**21IT1503-ARTIFICIAL INTELLIGENCE**

**UNIT-I INTELLIGENT AGENTS AND SEARCH TECHNIQUES**

## What is artificial intelligence?

- **Artificial Intelligence** is the branch of computer science concerned with making computers behave like humans.
- Major AI textbooks define artificial intelligence as "the study and design of intelligent agents," where an **intelligent agent** is a system that **perceives** its **environment** and **takes actions** which maximize its chances of success.
- **John McCarthy**, who coined the term in 1956, defines it as "the science and engineering of making intelligent machines, especially intelligent computer programs."
- The definitions of AI according to some text books are categorized into four approaches and are summarized in the table below :

| | |
|---|---|
| **Systems that think like humans**<br>"The exciting new effort to make computers think … machines with minds, in the full and literal sense."(Haugeland,1985) | **Systems that think rationally**<br>"The study of mental faculties through the use of computer models."<br>(Charniak and McDermont,1985) |
| **Systems that act like humans**<br>The art of creating machines that performs functions that require intelligence when performed by people."(Kurzweil,1990) | **Systems that act rationally**<br>"Computational intelligence is the study of the design of intelligent agents."(Poole et al.,1998) |

## Applications of Artificial Intelligence:

- **Autonomous planning and scheduling:**

  **A** hundred million miles from Earth, NASA's Remote Agent program became the first on-board autonomous planning program to control the scheduling of operations for a spacecraft (Jonsson *et* al., 2000). Remote Agent generated plans from high-level goals specified from the ground, and it monitored the operation of the spacecraft as the plans were executed-detecting, diagnosing, and recovering from problems as they occurred.

- **Game playing:**

  IBM's Deep Blue became the first computer program to defeat the world champion in a chess match when it bested Garry Kasparov by a score of 3.5 to 2.5 in an exhibition match (Goodman and Keene, 1997).

- **Autonomous control:**

  The ALVINN computer vision system was trained to steer a car to keep it following a lane. It was placed in CMU's NAVLAB computer-controlled minivan and used to navigate across the United States-for 2850 miles it was in control of steering the vehicle 98% of the time.

- **Diagnosis:**

  Medical diagnosis programs based on probabilistic analysis have been able to perform at the level of an expert physician in several areas of medicine.

- **Logistics Planning:**

  During the Persian Gulf crisis of 1991, U.S. forces deployed a Dynamic Analysis and Replanning Tool, DART (Cross and Walker, 1994), to do automated logistics planning and scheduling for transportation. This involved up to 50,000 vehicles, cargo, and people at a time, and had to account for starting points, destinations, routes, and conflict resolution among all parameters. The AI planning techniques allowed a plan to be generated in hours that would have taken weeks with older methods. The Defense Advanced Research Project Agency (DARPA) stated that this single application more than paid back DARPA's 30-year investment in AI.

- **Robotics:**

  Many surgeons now use robot assistants in microsurgery. HipNav (DiGioia *et* al., 1996) is a system that uses computer vision techniques to create a three-dimensional model of a patient's internal anatomy and then uses robotic control to guide the insertion of a hip replacement prosthesis.

- **Language understanding and problem solving:**

  PROVERB (Littman *et al.*, 1999) is a computer program that solves crossword puzzles better than most humans, using constraints on possible word fillers, a large database of past puzzles, and a variety of information sources including dictionaries and online databases such as a list of movies and the actors that appear in them.

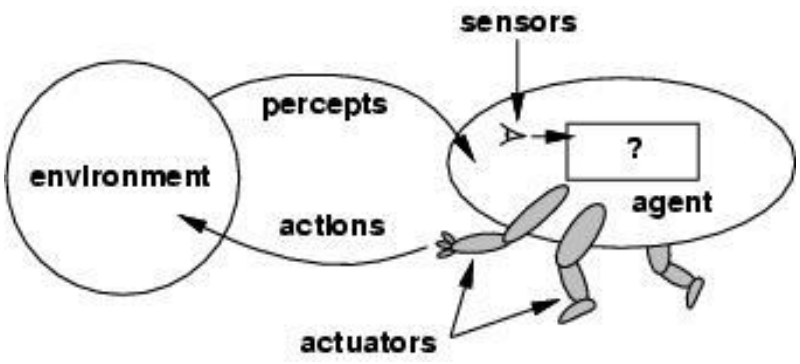
## AGENTS:

Rationality concept can be used to develop a smallest of design principle for building successful agents; these systems are reasonably called as Intelligent.

## Agents and environments:

An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and SENSOR acting upon that environment through **actuators.** This simple idea is illustrated in Figure.

- A human agent has eyes, ears, and other organs for sensors and hands, legs, mouth, and other body parts for actuators.
- A robotic agent might have cameras and infrared range finders for sensors and various motors for actuators.
- A software agent receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.




**Percept**

We use the term **percept** to refer to the agent's perceptual inputs at any given instant.

**Percept Sequence**

An agent's **percept sequence** is the complete history of everything the agent has ever perceived.

**Agent function**

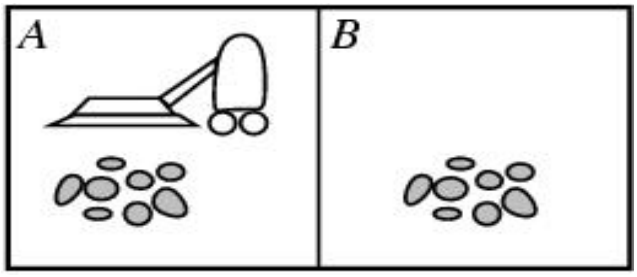
Mathematically speaking, we say that an agent's behavior is described by the **agent function** that maps any given percept sequence to an action.

$$f : \mathcal{P}^* \rightarrow \mathcal{A}$$

**Agent program**

- The agent function for an artificial agent will be implemented by an **agent program.**
- It is important to keep these two ideas distinct.
- The agent function is an abstract mathematical description;
- the agent program is a concrete implementation, running on the agent architecture.

- To illustrate these ideas, we will use a very simple example-the vacuum-cleaner world shown in Figure.
- This particular world has just two locations: squares A and B.
- The vacuum agent perceives which square it is in and whether there is dirt in the square.
- It can choose to move left, move right, suck up the dirt, or do nothing.
- One very simple agent function is the following:
- if the current square is dirty, then suck, otherwise,
- it move to the other square.
- A partial tabulation of this agent function is shown in Figure.




**Agent function**

| **Percept Sequence** | **Action** |
|---|---|
| [A, Clean] | Right |
| [A, Dirty] | Suck |
| [B, Clean] | Left |
| [B, Dirty] | Suck |
| [A, Clean], [A, Clean] | Right |
| [A, Clean], [A, Dirty] | Suck |
| ….. | ….. |
| | |

**Agent program**

**function** Reflex-VACUUM-AGENT ([locations, status]) **returns an action**

**if** status = Dirty **then return** Suck
**else if** location = A **then return** Right
**elseif** location = B **then return** Left

## Good Behavior: The concept of Rationality

- A **rational agent** is one that does the right thing-conceptually speaking; every entry in the table for the agent function is filled out correctly.
- Obviously, doing the right thing is better than doing the wrong thing.
- The right action is the one that will cause the agent to be most successful.

**Performance measures**

- **A performance measure** embodies the **criterion for success** of an agent's behavior.
- When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives.
- This sequence of actions causes the environment to go through a sequence of states.
- If the sequence is desirable, then the agent has performed well.

**Rationality**

What is rational at any given time depends on four things:

- The performance measure that defines the criterion of success.
- The agent's prior knowledge of the environment.
- The actions that the agent can perform.
- The agent's percept sequence to date.
- This leads to a **definition of a rational agent:**

**For each possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.**

**Omniscience, learning, and autonomy**

- An **omniscient agent** knows the actual outcome of its actions and can act accordingly; but omniscience is impossible in reality.
- Doing actions in order to modify future percepts-sometimes called **information gathering**-is an important part of rationality.
- Our definition requires a rational agent not only to gather information, but also to **learn** as much as possible from what it perceives.
- To the extent that an agent relies on the prior knowledge of its designer rather than on its own percepts, we say that the agent lacks autonomy.
- A rational agent should be **autonomous**-it should learn what it can to compensate for partial or incorrect prior knowledge.

**Task environments**

- We must think about **task environments,** which are essentially the "**problems**" to which rational agents are the "**solutions**."

**Specifying the task environment**

- The rationality of the simple vacuum-cleaner agent, needs specification of
  - ✓ the performance measure
  - ✓ the environment
  - ✓ the agent's actuators
  - ✓ Sensors.

## PEAS

- All these are grouped together under the heading of the **task environment.**
- We call this the **PEAS** (Performance, Environment, Actuators, Sensors) description.
- In designing an agent, the first step must always be to specify the task environment as fully as possible.
- The following table shows PEAS description of the task environment for an automated taxi.

| **Agent Type** | **Performance Measure** | **Environments** | **Actuators** | **Sensors** |
|---|---|---|---|---|
| Taxi driver | Safe: fast, legal, comfortable trip, maximize profits | Roads,other traffic,pedestrians, customers | Steering,accelerator, brake, Signal,horn,display | Cameras,sonar, Speedometer,GPS, Odometer,engine sensors,keyboards, accelerometer |

- The following table shows PEAS description of the task environment for some other agent type.

| Agent Type | Performance Measure | Environment | Actuators | Sensors |
|---|---|---|---|---|
| Medical diagnosis system | Healthy patient, minimize costs, lawsuits | Patient, hospital, staff | Display questions, tests, diagnoses, treatments, referrals | Keyboard entry of symptoms, findings, patient's answers |
| Satellite image analysis system | Correct image categorization | Downlink from orbiting satellite | Display categorization of scene | Color pixel arrays |
| Part-picking robot | Percentage of parts in correct bins | Conveyor belt with parts; bins | Jointed arm and hand | Camera, joint angle sensors |
| Refinery controller | Maximize purity, yield, safety | Refinery, operators | Valves, pumps, heaters, displays | Temperature, pressure, chemical sensors |
| Interactive English tutor | Maximize student's score on test | Set of students, testing agency | Display exercises, suggestions, corrections | Keyboard entry |

**Properties of task environments**

- **Fully observable vs. partially observable**
- **Deterministic vs. stochastic**
- **Episodic vs. sequential**
- **Static vs. dynamic**
- **Discrete vs. continuous**
- **Single agent vs. multi agent**

**Fully observable** vs. **partially observable.**

- If an agent's sensors give it access to the complete state of the environment at each point in time, then we say that the task environment is fully observable.
- A task environment is effectively fully observable if the sensors detect all aspects that are *relevant* to the choice of action;
- An environment might be partially observable because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data.

**Deterministic** vs. **stochastic.**

- If the next state of the environment is completely determined by the current state and the action executed by the agent, then we say the environment is deterministic;
- Otherwise, it is stochastic.

**Episodic** vs. **sequential**

- In an **episodic task environment**, the agent's experience is divided into atomic episodes.
- Each episode consists of the agent perceiving and then performing a single action. Crucially, the next episode does not depend on the actions taken in previous episodes.
- For example, an agent that has to spot defective parts on an assembly line bases each decision on the current part, regardless of previous decisions;
- In **sequential environments**, on the other hand, the current decision Could affect all future decisions.
- Chess and taxi driving are sequential:

**Discrete** vs. **continuous.**

- The discrete/continuous distinction can be applied to the *state* of the environment, to the way *time* is handled, and to the *percepts* and *actions* of the agent.
- For example, a discrete-state environment such as a chess game has a finite number of distinct states.
- Chess also has a discrete set of percepts and actions.
- Taxi driving is a continuous- state and continuous-time problem:
- The speed and location of the taxi and of the other vehicles sweep through a range of continuous values and do so smoothly over time.
- Taxi-driving actions are also continuous (steering angles, etc.)

**Single agent** vs. **multi agent.**

- An agent solving a crossword puzzle by itself is clearly in a single-agent environment,
- Where as an agent playing chess is in a two-agent environment.
- Multi agent is further classified in to two ways

  - ✓ Competitive multi agent environment
  - ✓ Cooperative multi agent environment

## Agent programs

- The job of Artificial Intelligence is to design the agent program that implements the agent function mapping percepts to actions
- The agent program will run in an architecture
- An architecture is a computing device with physical sensors and actuators
- Where Agent is combination of Program and Architecture

**Agent = Program + Architecture**

- An agent program takes the current percept as input while the agent function takes the entire percept history
- Current percept is taken as input to the agent program because nothing more is available from the environment
- The following TABLE-DRIVEN_AGENT program is invoked for each new percept and returns an action each time

**Function** TABLE-DRIVEN_AGENT (percept) **returns** an action

**static**: percepts, a sequence initially empty
table, a table of actions, indexed by percept sequence

append percept to the end of percepts
action ← LOOKUP(percepts, table)
**return** action

**Drawbacks:**

- **Table lookup** of percept-action pairs defining all possible condition-action rules necessary to interact in an environment
- **Problems**
  - Too big to generate and to store (Chess has about 10^120 states, for example)
  - No knowledge of non-perceptual parts of the current state
  - Not adaptive to changes in the environment; requires entire table to be updated if changes occur
  - Looping: Can't make actions conditional

- Take a long time to build the table
- No autonomy
- Even with learning, need a long time to learn the table entries

## Some Agent Types

- **Table-driven agents**
  - use a percept sequence/action table in memory to find the next action. They are implemented by a (large) **lookup table**.
- **Simple reflex agents**
  - are based on **condition-action rules**, implemented with an appropriate production system. They are stateless devices which do not have memory of past world states.
- **Agents with memory**
  - have **internal state**, which is used to keep track of past states of the world.
- **Agents with goals**
  - are agents that, in addition to state information, have **goal information** that describes desirable situations. Agents of this kind take future events into consideration.
- **Utility-based agents**
  - base their decisions on **classic axiomatic utility theory** in order to act rationally.

## Kinds of Agent Programs

- The following are the agent programs,
  - Simple reflex agents
  - Mode-based reflex agents
  - Goal-based reflex agents
  - Utility-based agents

### Simple Reflex Agent

- The simplest kind of agent is the **simple reflex agent.**
- These agents select actions on the basis of the *current* percept, ignoring the rest of the percept history.
- For example, the vacuum agent whose agent function is tabulated is given below,
- a simple reflex agent, because its decision is based only on the current location and on whether that contains dirt.
- Select action on the basis of *only the current* percept. E.g. the vacuum-agent
- Large reduction in possible percept/action situations (next page).
- Implemented through *condition-action rules*
- If dirty then suck

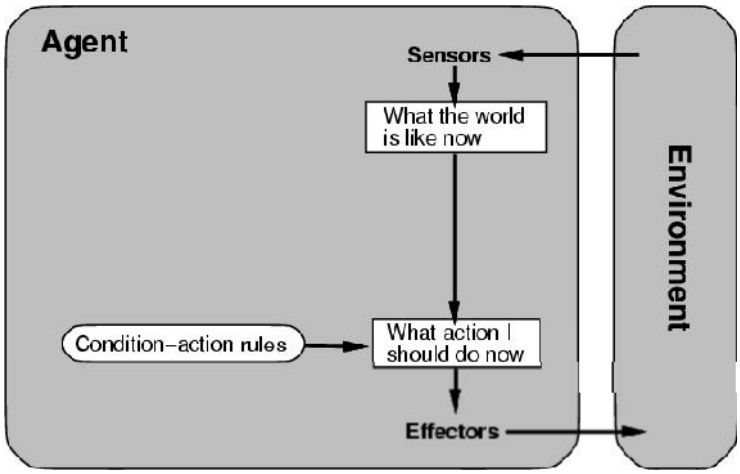
### A Simple Reflex Agent: Schema

- Schematic diagram of a simple reflex agent.
- The following simple reflex agents, acts according to a rule whose condition matches the current state, as defined by the percept

**function** SIMPLE-REFLEX-AGENT(*percept*) **returns** an action
**static**: *rules*, a set of condition-action rules
*state* ← INTERPRET-

INPUT(*percept*)

*rule* ← RULE-MATCH(*state*, *rule*)
*action* ← RULE-ACTION[*rule*] return *action*




- The agent program for a simple reflex agent in the two-state vacuum environment.

*function REFLEX-VACUUM-AGENT ([location, status]) return an action*
*if status == Dirty then return Suck*
*else if location == A then return Right*
*else if location == B then return Left*

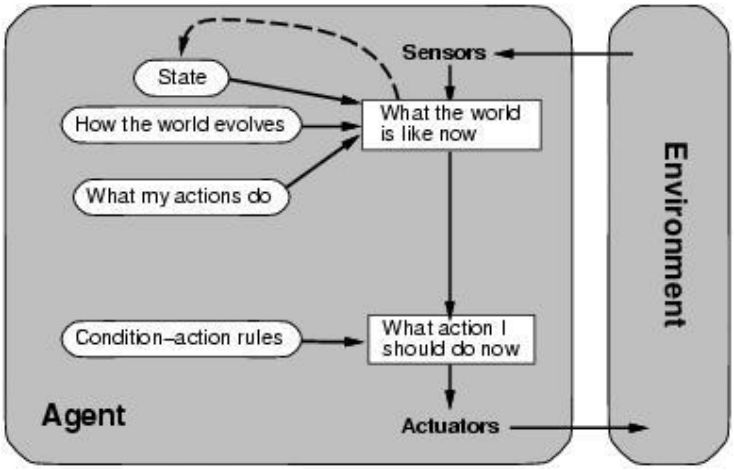
## Characteristics

- Only works if the environment is fully observable.
- Lacking history, easily get stuck in infinite loops
- One solution is to randomize actions

## Model-based reflex agents

- The most effective way to handle partial observability is for the agent to *keep track of the part of the world it can't see now.*
- That is, the agent should maintain some sort of **internal state** that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state.
- Updating this internal state information as time goes by requires two kinds of knowledge to be encoded in the agent program.
- First, we need some information about how the world evolves independently of the agent
- For example, that an overtaking car generally will be closer behind than it was a moment ago.
- Second, we need some information about how the agent's own actions affect the world

- For example, that when the agent turns the steering wheel clockwise, the car turns to the right or that after driving for five minutes northbound on the freeway one is usually about five miles north of where one was five minutes ago.
- This knowledge about "how the world working - whether implemented in simple Boolean circuits or in complete scientific theories-is called a **model** of the world.
- An agent that uses such a MODEL-BASED model is called a **model-based agent.**
- Schematic diagram of A model based reflex agent

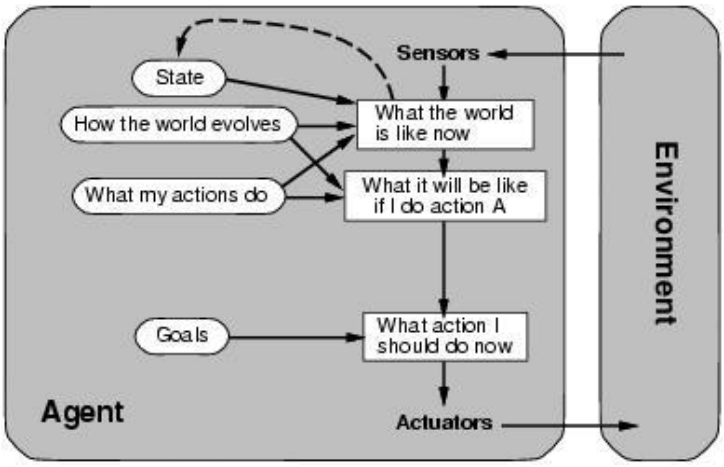



- Model based reflex agent. It keeps track of the current state of the world using an internal model. It then chooses an action in the same way as the reflex agent.

**function** REFLEX-AGENT-WITH-STATE(*percept*) **returns** an action
**static**: *rules*, a set of condition-action rules
*state*, a description of the current world state
*action*, the most recent action.
*state* ← VPDATE-STATE(*state*, *action*, *percept*)
*rule* ← RVLE-MATCH(*state*, *rule*)
*action* ← RVLE-ACTION[*rule*] return *action*

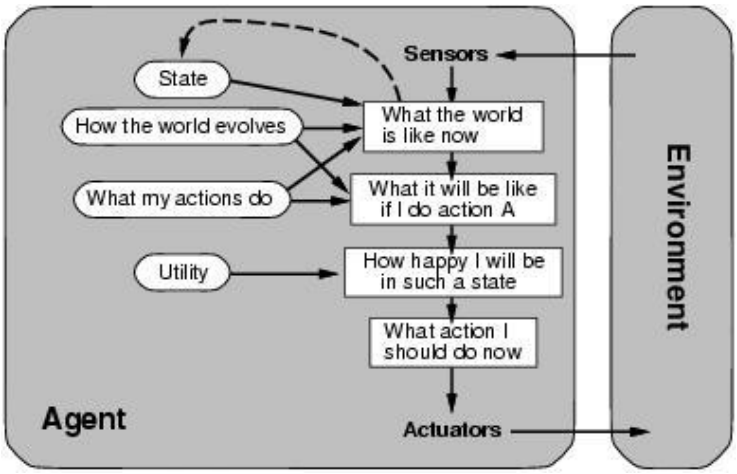
## Goal-based agents

- Knowing about the current state of the environment is not always enough to decide what to do.
- For example, at a road junction, the taxi can turn left, turn right, or go straight on. The correct decision depends on where the taxi is trying to get to.
- In other words, as well as a current state description, the agent needs some sort of **goal** information that describes situations that are desirable.
- For example, being at the passenger's destination.
- The agent program can combine this with information about the results of possible actions (the same information as was used to update internal state in the reflex agent) in order to choose actions that achieve the goal.
- Schematic diagram of the goal-based agent's structure.



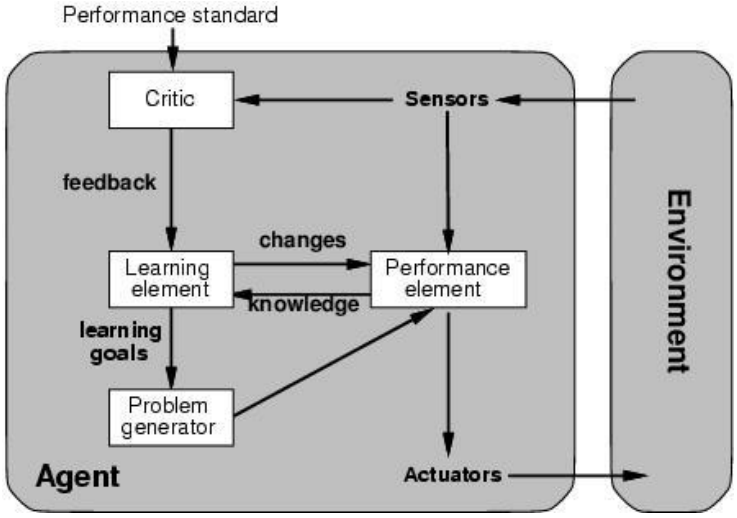

**Utility-based agents**

- Goals alone are not really enough to generate high-quality behavior in most environments.
- For example, there are many action sequences that will get the taxi to its destination (thereby achieving the goal) but some are quicker, safer, more reliable, or cheaper than others.
- Goals just provide a crude binary distinction between "happy" and "unhappy" states, whereas a more general **performance measure** should allow a comparison of different world states according to exactly how happy they would make the agent if they could be achieved. Because "happy" does not sound very scientific, the customary terminology is to say that if one world state is preferred to another, then it has higher **utility** for the agent.
- Schematic diagram of a utility-based agents
- It uses a model of the world, along with a utility function that measures its preferences among states of the world.
- Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.
- Certain goals can be reached in different ways
    - Some are better, have a higher utility
- Vtility function maps a (Sequence of) state(S) onto a real number.
- Improves on goal:
    - Selecting between conflicting goals
    - Select appropriately between several goals based on likelihood of Success




## Learning Agent

Schematic diagram of Learning Agent




- All agents can improve their performance through learning.
- A learning agent can be divided into four conceptual components, as,
    - Learning element
    - Performance element
    - Critic
    - Problem generator
- The most important distinction is between the **learning element**, which is responsible for making improvements,
- The **performance element**, which is responsible for selecting external actions.
- The performance element is what we have previously considered to be the entire agent: it takes in percepts and decides on actions.
- The learning element uses feedback from the **critic** on how the agent is doing and determines how the performance element should be modified to do better in the future.
- The last component of the learning agent is the **problem generator**.

- It is responsible for suggesting actions that will lead to new and **informative experiences**. But if the agent is willing to explore a little, it might discover much better actions for the long run.
- The problem generator's job is to suggest these **exploratory actions**. This is what scientists do when they carry out experiments.

### Summary: Intelligent Agents

- An **agent** perceives and acts in an environment, has an architecture, and is implemented by an agent program.
- Task environment - **PEAS (P**erformance**, E**nvironment, **A**ctuators, **S**ensors**)**
- The most challenging environments are inaccessible, nondeterministic, dynamic, and continuous.
- An **ideal agent** always chooses the action which maximizes its expected performance, given its percept sequence so far.
- An **agent program** maps from percept to action and updates internal state.
  - **Reflex agents** respond immediately to percepts.
    - simple reflex agents
    - model-based reflex agents
  - **Goal-based agents** act in order to achieve their goal(s).
  - **Utility-based agents** maximize their own utility function.
- All agents can improve their performance through **learning.**

## Problem Formulation

- An important aspect of intelligence is *goal-based* problem solving.
- The solution of many problems can be described by finding a sequence of actions that lead to a desirable goal.
- Each action changes the *state* and the aim is to find the sequence of actions and states that lead from the initial (start) state to a final (goal) state.
- **A well-defined problem can be described by:**
  - **Initial state**
  - **Operator or successor function** - for any state x returns s(x), the set of states reachable from x with one action
  - **State space** - all states reachable from initial by any sequence of actions
  - **Path** - sequence through state space
  - **Path cost** - function that assigns a cost to a path. Cost of a path is the sum of costs of individual actions along the path
  - **Goal test** - test to determine if at goal state
- What is **Search**?
- Search is the systematic examination of states to find path from the start/root state to the goal state.
- The set of possible states, together with *operators* defining their connectivity constitute the *search space*.
- The output of a search algorithm is a solution, that is, a path from the initial state to a state that satisfies the goal test.

# Problem-solving agents

- A Problem solving agent is a goal-based agent.
- It decides what to do by finding sequence of actions that lead to desirable states.
- The agent can adopt a goal and aim at satisfying it.
- To illustrate the agent's behavior
- For example where our agent is in the city of Arad, which is in Romania. The agent has to adopt a goal of getting to Bucharest.
- Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving.
- The agent's task is to find out which sequence of actions will get to a goal state.
- Problem formulation is the process of deciding what actions and states to consider given a goal.

Example: Route finding problem
On holiday in Romania : currently in Arad.
Flight leaves tomorrow from Bucharest
**Formulate goal**: be in Bucharest

**Formulate problem**:
**states**: various cities
**actions**: drive between cities

**Find solution**:
sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

- Goal formulation and problem formulation A

**problem** is defined by four items:
**initial state** e.g., "at Arad"
**successor function** S(x) = set of action-state pairs
e.g., S(Arad) = {[Arad -> Zerind;Zerind],….} **goal**
**test**, can be
explicit, e.g., x = at Bucharest"
implicit, e.g., NoDirt(x)
**path cost** (additive)
e.g., sum of distances, number of actions executed, etc.
c(x; a; y) is the step cost, assumed to be >= 0

A **solution** is a sequence of actions leading from the initial state to a goal state.

# Search

- An agent with several immediate options of unknown value can decide what to do by examining different possible sequences of actions that leads to the states of known value,and then choosing the best sequence.
- The process of looking for sequences actions from the current state to reach the goal state is called **search.**

- The **search algorithm** takes a **problem** as **input** and returns a **solution** in the form of **action sequence.**
- Once a solution is found,the **execution phase** consists of carrying out the recommended action.
- The following shows a simple "formulate,search,execute" design for the agent.
- Once solution has been executed, the agent will formulate a new goal.
- It first formulates a **goal** and a **problem**,searches for a sequence of actions that would solve a problem,and executes the actions one at a time.

**function** SIMPLE-PROBLEM-SOLVING-AGENT( *percept*) **returns** an action
**inputs** : *percept*, a percept
**static**: *seq*, an action sequence, initially empty
*state*, some description of the current world state
*goal*, a goal, initially null
*problem*, a problem formulation
state VPDATE-STATE(*state*, *percept*)
**if** seq is empty **then do**
*goal* ← FORMVLATE-GOAL(*state*)
*problem* ← FORMVLATE-PROBLEM(*state*, *goal*)
*seq* ← SEARCH( *problem*)
*action* ← FIRST(*seq*);
*seq* ← REST(seq)
**return** *action*

- The agent design assumes the Environment is

- **Static**: The entire process carried out without paying attention to changes that might be occurring in the environment.
- **Observable :** The initial state is known and the agent's sensor detects all aspects that are relevant to the choice of action
- **Discrete :** With respect to the state of the environment and percepts and actions so that alternate courses of action can be taken
- **Deterministic:** The next state of the environment is completely determined by the current state and the actions executed by the agent. Solutions to the problem are single sequence of actions

An agent carries out its plan with eye closed. This is called an open loop system because ignoring the percepts breaks the loop between the agent and the environment.

## Well-defined problems and solutions

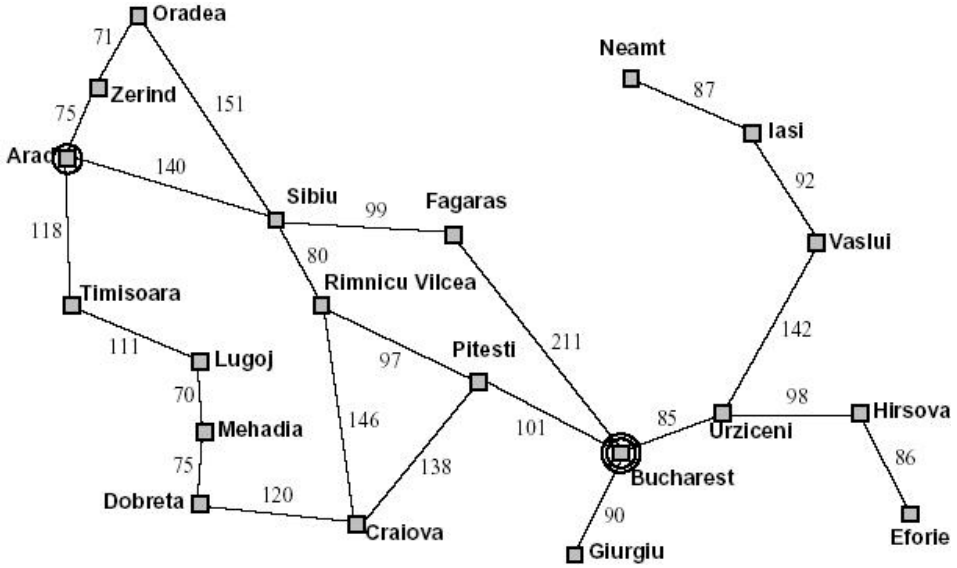
A **problem** can be formally defined by **four components**:

- The **initial state** that the agent starts in . The initial state for our agent of example problem is described by *In(Arad)*
- A **Successor Function** returns the possible **actions** available to the agent.
- Given a state x,SVCCESSOR-FN(x) returns a set of {action,successor} ordered pairs where each action is one of the legal actions in state x,and each successor is a state that can be reached from x by applying the action.

- For example, from the state In(Arad),the successor function for the Romania problem would return

{ [Go(Sibiu),In(Sibiu)],[Go(Timisoara),In(Timisoara)],[Go(Zerind),In(Zerind)] }

- **State Space**: The set of all states reachable from the initial state. The state space forms a graph in which the nodes are states and the arcs between nodes are actions.
- A **path** in the state space is a sequence of states connected by a sequence of actions.
- The **goal test** determines whether the given state is a goal state.
- A **path cost** function assigns numeric cost to each action.
- For the Romania problem the cost of path might be its length in kilometers.
- The **step cost** of taking action a to go from state x to state y is denoted by c(x,a,y). It is assumed that the step costs are non negative.
- A **solution** to the problem is a path from the initial state to a goal state.
- An **optimal solution** has the lowest path cost among all solutions.




A simplified Road Map of part of Romania

## Advantages:

- They are easy enough because they can be carried out without further search or planning
- The choice of a good abstraction thus involves removing as much details as possible while retaining validity and ensuring that the abstract actions are easy to carry out.

# EXAMPLE PROBLEMS

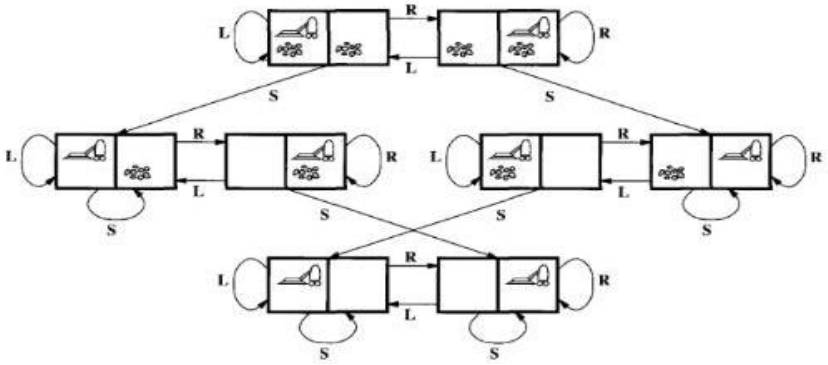
- The problem solving approach has been applied to a vast array of task environments. Some best known problems are summarized below.
- They are distinguished as toy or real-world problems
  - A T**oy problem** is intended to illustrate various problem solving methods. It can be easily used by different researchers to compare the performance of algorithms.
  - A R**eal world problem** is one whose solutions people actually care about.

## TOY PROBLEMS

**Vacuum World Example**

- **States**: The agent is in one of two locations.,each of which might or might not contain dirt. Thus there are $2 \times 2^2 = 8$ possible world states.
- **Initial state**: Any state can be designated as initial state.
- **Successor function** : This generates the legal states that results from trying the three actions (left, right, suck). The complete state space is shown in figure 2.3
- **Goal Test** : This tests whether all the squares are clean.
- **Path test** : Each step costs one ,so that the the path cost is the number of steps in the path.

**Vacuum World State Space**

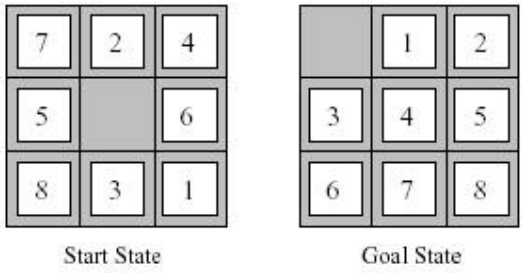



The state space for the vacuum world.
Arcs denote actions: L = Left,R = Right,S = Suck

## 8-puzzle:

- An 8-puzzle consists of a 3x3 board with eight numbered tiles and a blank space.
- A tile adjacent to the blank space can slide into the space. The object is to reach the specific goal state ,as shown in figure

**Example: The 8-puzzle**




A typical instance of 8-puzzle.

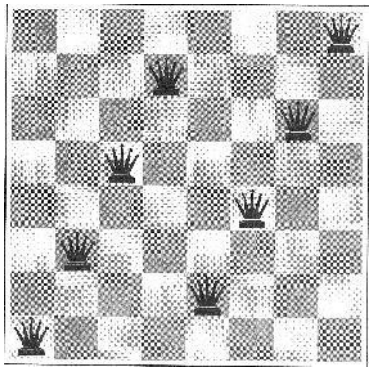
The problem formulation is as follows :

- **States** : A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state** : Any state can be designated as the initial state. It can be noted that any given goal can be reached from exactly half of the possible initial states.
- **Successor function** : This generates the legal states that result from trying the four actions (blank moves Left,Right,Vp or down).
- **Goal Test** : This checks whether the state matches the goal configuration shown in figure 2.4.(Other goal configurations are possible)
- **Path cost**: Each step costs 1,so the path cost is the number of steps in the path.
- **The 8-puzzle** belongs to the family **of sliding-block puzzles**, which are often used as test problems for new search algorithms in AI.
- This general class is known as NP-complete.
- The **8-puzzle** has 9!/2 = 181,440 reachable states and is easily solved.
- The **15 puzzle** ( 4 x 4 board ) has around 1.3 trillion states, an the random instances can be solved optimally in few milli seconds by the best search algorithms.
- The **24-puzzle** (on a 5 x 5 board) has around $10^{25}$ states ,and random instances are still quite difficult to solve optimally with current machines and algorithms.

# 8-queens problem

- The goal of 8-queens problem is to place 8 queens on the chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column or diagonal).
- The following figure shows an attempted solution that fails: the queen in the right most column is attacked by the queen at the top left.
- An **Incremental formulation** involves operators that augments the state description,starting with an empty state.for 8-queens problem,this means each action adds a queen to the state.
- A **complete-state formulation** starts with all 8 queens on the board and move them around.

In either case the path cost is of no interest because only the final state counts.



8-queens problem

- The first incremental formulation one might try is the following :
    - **States** : Any arrangement of 0 to 8 queens on board is a state.
    - **Initial state** : No queen on the board.
    - **Successor function** : Add a queen to any empty square.
    - **Goal Test** : 8 queens are on the board,none attacked.
- In this formulation, we have $64.63\ldots57 = 3 \times 10^{14}$ possible sequences to investigate.
- A better formulation would prohibit placing a queen in any square that is already attacked. :
    - **States** : Arrangements of n queens ( 0 <= n < = 8 ) ,one per column in the left most columns ,with no queen attacking another are states.
    - **Successor function** : Add a queen to any square in the left most empty column such that it is not attacked by any other queen.
- This formulation reduces the 8-queen state space from $3 \times 10^{14}$ to just 2057,and solutions are easy to find.
- For the 100 queens the initial formulation has roughly $10^{400}$ states whereas the improved formulation has about $10^{52}$ states.
- This is a huge reduction, but the improved state space is still too big for the algorithms to handle.

# REAL WORLD PROBLEMS

- A real world problem is one whose solutions people actually care about.
- They tend not to have a single agreed upon description, but attempt is made to give general flavor of their formulation,
- The following are the some real world problems,
    - Route Finding Problem
    - Touring Problems
    - Travelling Salesman Problem
    - Robot Navigation

## ROUTE-FINDING PROBLEM

- Route-finding problem is defined in terms of specified locations and transitions along links between them.
- Route-finding algorithms are used in a variety of applications, such as routing in computer networks, military operations planning, and air line travel planning systems.

## AIRLINE TRAVEL PROBLEM

The **airline travel problem** is specifies as follows **:**

- **States :** Each is represented by a location(e.g.,an airport) and the current time.
- **Initial state :** This is specified by the problem.
- **Successor function :** This returns the states resulting from taking any scheduled flight(further specified by seat class and location),leaving later than the current time plus the within-airport transit time,from the current airport to another.
- **Goal Test :** Are we at the destination by some prespecified time?
- **Path cost :** This depends upon the monetary cost,waiting time,flight time,customs and immigration procedures,seat quality,time of dat,type of air plane,frequent-flyer mileage awards, and so on.

## TOURING PROBLEMS

- **Touring problems** are closely related to route-finding problems,but with an important difference.
- Consider for example, the problem, "Visit every city at least once" as shown in Romania map.
- As with route-finding the actions correspond to trips between adjacent cities. The state space, however,is quite different.

  - **Initial state** would be "In Bucharest; visited{Bucharest}".
  - **Intermediate state** would be "In Vaslui; visited {Bucharest,Vrziceni,Vaslui}".
  - **Goal test** would check whether the agent is in Bucharest and all 20 cities have been visited.

## THE TRAVELLING SALESPERSON PROBLEM (TSP)

- ✓ TSP is a touring problem in which each city must be visited exactly once.
- ✓ The aim is to find the shortest tour. The problem is known to be **NP-hard**.
- ✓ Enormous efforts have been expended to improve the capabilities of TSP algorithms.
- ✓ These algorithms are also used in tasks such as planning movements of **automatic circuit-board drills** and of **stocking machines** on shop floors.
- ✓

**VLSI layout**

A **VLSI layout** problem requires positioning millions of components and connections on a chip to minimize area ,minimize circuit delays,minimize stray capacitances,and maximize manufacturing yield. The layout problem is split into two parts : **cell layout** and **channel routing**.

**ROBOT navigation**

**ROBOT navigation** is a generalization of the route-finding problem. Rather than a discrete set of routes,a robot can move in a continuous space with an infinite set of possible actions and states. For a circular Robot moving on a flat surface,the space is essentially two-dimensional.

When the robot has arms and legs or wheels that also must be controlled,the search space becomes multi-dimensional. Advanced techniques are required to make the search space finite.

**AUTOMATIC ASSEMBLY SEQUENCING**

The example includes assembly of intricate objects such as electric motors. The aim in assembly problems is to find the order in which to assemble the parts of some objects. If the wrong order is choosen, there will be no way to add some part later without undoing somework already done.

Another important assembly problem is protein design, in which the goal is to find a sequence of

Amino acids that will be fold into a three-dimensional protein with the right properties to cure some disease.

**INTERNET SEARCHING**

In recent years there has been increased demand for software robots that perform Internet searching, looking for answers to questions, for related information, or for shopping deals.

The searching techniques consider internet as a graph of nodes (pages) connected by links.

**MEASURING PROBLEM-SOLVING PERFORMANCE**

- ✓ The output of problem-solving algorithm is either failure or a solution. (Some algorithms might struck in an infinite loop and never return an output.)
- ✓ The algorithm's performance can be measured in four ways :
    - **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
    - **Optinality** : Does the strategy find the optimal solution
    - **Time complexity:** How long does it take to find a solution?
    - **Space complexity:** How much memory is needed to perform the search?

# UNINFORMED SEARCH STRATGES

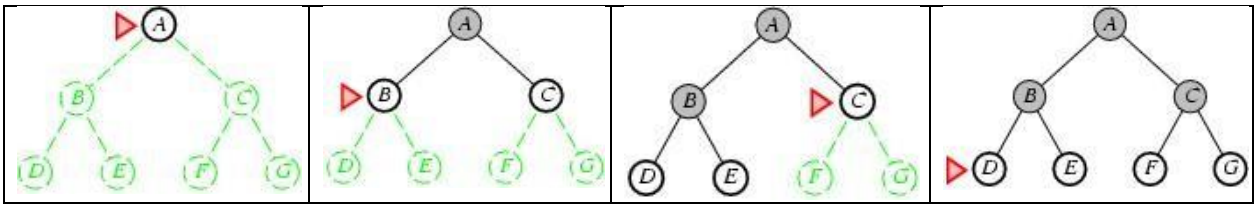
- ✓ **Uninformed Search Strategies** have no additional information about states beyond that provided in the **problem definition**.
- ✓ **Strategies** that know whether one non goal state is "more promising" than another are called I**nformed search or heuristic search** strategies.

There are five uninformed search strategies as given below.

- Breadth-first search
- Vniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search
- Bidirectional Search

## Breadth-first search

- ✓ Breadth-first search is a simple strategy in which the root node is expanded first, then all successors of the root node are expanded next, then their successors, and so on.
- ✓ In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
- ✓ Breath-first-search is implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out(FIFO) queue, assuring that the nodes that are visited first will be expanded first.
- ✓ In otherwards, calling TREE-SEARCH (problem,FIFO-QVEVE()) results in breadth-first-search.
- ✓ The FIFO queue puts all newly generated successors at the end of the queue, which means that Shallow nodes are expanded before deeper nodes.




Breadth-first search on a simple binary tree. At each stage ,the node to be expanded next is indicated by a marker.

**Properties of breadth-first-search**

Complete?? Yes (if $b$ is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in $d$

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? No, unless step costs are constant

Space is the big problem; can easily generate nodes at 100MB/sec
so 24hrs = 8640GB.

## Time and Memory Requirements for BFS – $O(b^{d+1})$

**Example:**
- b = 10
- 10000 nodes/second
- each node requires 1000 bytes of storage

| Depth | Nodes | Time | Memory |
|---|---|---|---|
| 2 | 1100 | .11 sec | 1 meg |
| 4 | 111,100 | 11 sec | 106 meg |
| 6 | $10^7$ | 19 min | 10 gig |
| 8 | $10^9$ | 31 hrs | 1 tera |
| 10 | $10^{11}$ | 129 days | 101 tera |
| 12 | $10^{13}$ | 35 yrs | 10 peta |
| 14 | $10^{15}$ | 3523 yrs | 1 exa |

Time and memory requirements for breadth-first-search.

**Time complexity for BFS**

- ✓ Assume every state has b successors.
- ✓ The root of the search tree generates b nodes at the first level,each of which generates b more nodes,for a total of $b^2$ at the second level.
- ✓ Each of these generates b more nodes,yielding $b^3$ nodes at the third level,and so on.
- ✓ Now suppose,that the solution is at depth d.
- ✓ In the worst case,we would expand all but the last node at level d,generating $b^{d+1}$ - b nodes at level d+1.
- ✓ Then the total number of nodes generated is
  $b + b^2 + b^3 + \ldots + b^d + (bd^{+1} + b) = O(b^{d+1)}$.
- ✓ Every node that is generated must remain in memory,because it is either part of the fringe or is an ancestor of a fringe node.
- ✓ The space compleity is,therefore ,the same as the time complexity

## UNIFORM-COST SEARCH

- ✓ Instead of expanding the shallowest node,**uniform-cost search** expands the node n with the lowest path cost.
- ✓ uniform-cost search does not care about the number of steps a path has,but only about their total cost.

**Properties of Uniform-cost-search:**

Expand least-cost unexpanded node

Implementation:
$fringe$ = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost $\geq \epsilon$

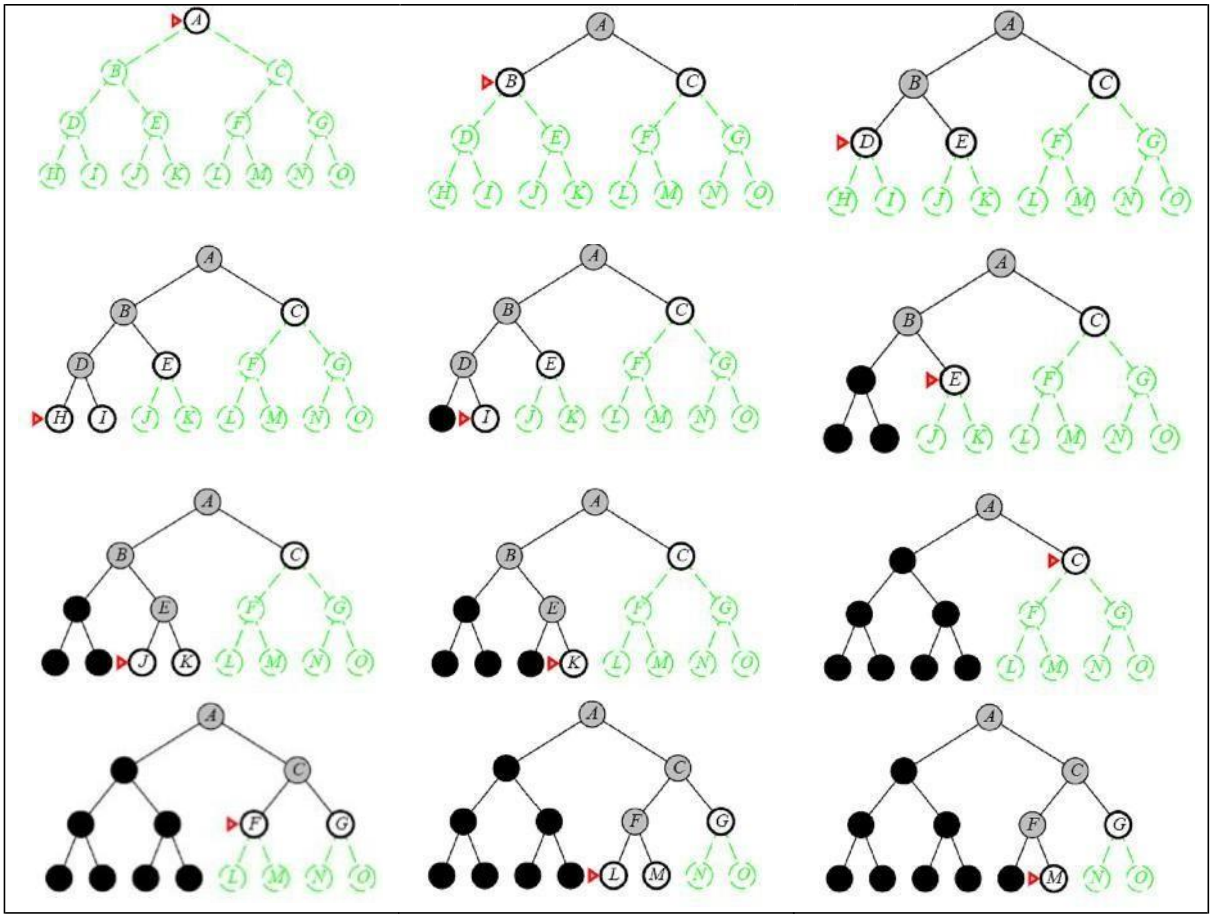
Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where $C^*$ is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

Optimal?? Yes—nodes expanded in increasing order of $g(n)$

## DEPTH-FIRST-SEARCH

- Depth-first-search always expands the deepest node in the current fringe of the search tree.
- The progress of the search is illustrated in figure.
- The search proceeds immediately to the deepest level of the search tree,where the nodes have no successors.
- As those nodes are expanded,they are dropped from the fringe,
- so then the search "backs up" to the next shallowest node that still has unexplored successors.
- This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue,also known as a stack.
- Depth-first-search has very modest memory requirements.
- It needs to store only a single path from the root to a leaf node,along with the remaining unexpanded sibling nodes for each node on the path.
- Once the node has been expanded,it can be removed from the memory,as soon as its descendants have been fully explored.
- For a state space with a branching factor b and maximum depth m,depth-first-search requires storage of only bm + 1 nodes.



Depth-first-search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from the memory; these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

**Drawback of Depth-first-search**

The drawback of depth-first-search is that it can make a wrong choice and get stuck going down very long(or even infinite) path when a different choice would lead to solution near the root of the search tree.

For example, depth-first-search will explore the entire left subtree even if node C is a goal node.

**BACKTRACKING SEARCH**

A variant of depth-first search called backtracking search uses less memor y and only one successor is generated at a time rather than all successors.; Only O(m) memory is needed rather than $O(b^m)$

## DEPTH-LIMITED-SEARCH

- The problem of unbounded trees can be alleviated by supplying depth-first-search with a pre-determined depth limit l.
- That is,nodes at depth l are treated as if they have no successors.
- This approach is called **depth-limited-search**.
- The depth limit solves the infinite path problem.
- Depth limited search will be nonoptimal if we choose l > d. Its time complexity is $O(b^l)$ and its space compleiy is O(bl).
- Depth-first-search can be viewed as a special case of depth-limited search with l = oo
- Sometimes,depth limits can be based on knowledge of the problem.
- For,example,on the map of Romania there are 20 cities.
- Therefore,we know that if there is a solution.,it must be of length 19 at the longest,So l = 10 is a possible choice.
- However,it oocan be shown that any city can be reached from any other city in at most 9 steps.
- This number known as the **diameter** of the state space,gives us a better depth limit.
- Depth-limited-search can be implemented as a simple modification to the general tree-search algorithm or to the recursive depth-first-search algorithm.
- The pseudocode for recursive depth-limited-search is shown.
- It can be noted that the above algorithm can terminate with two kinds of failure : the standard *failure* value indicates no solution; the *cutoff* value indicates no solution within the depth limit.
- Depth-limited search = depth-first search with depth limit l, returns cut off if any path is cut off by depth limit
- Recursive implementation of Depth-limited-search:

```
function Depth-Limited-Search( problem, limit) returns a solution/fail/cutoff
return Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)
function Recursive-DLS(node, problem, limit) returns solution/fail/cutoff
cutoff-occurred? ← false
if Goal-Test(problem,State[node]) then return Solution(node)
else if Depth[node] = limit then return cutoff
else for each successor in Expand(node, problem) do
result ← Recursive-DLS(successor, problem, limit) if
result = cutoff then cutoff_occurred? ← true
else if result not = failure then return result
if cutoff_occurred? then return cutoff else return failure
```

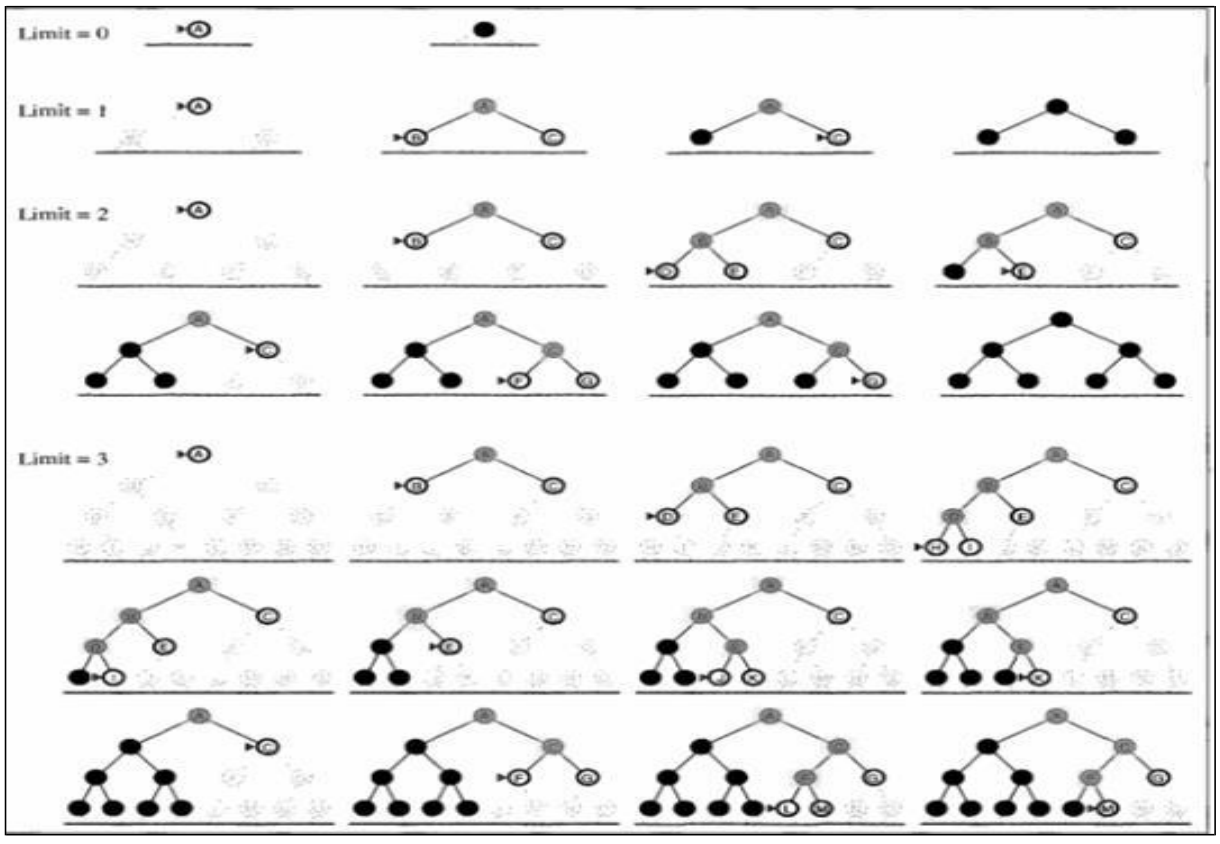
## ITERATIVE DEEPENING DEPTH-FIRST SEARCH

- Iterative deepening search (or iterative-deepening-depth-first-search) is a general strategy often used in combination with depth-first-search,that finds the better depth limit.
- It does this by gradually increasing the limit - first 0,then 1,then 2, and so on - until a goal is found.
- This will occur when the depth limit reaches d,the depth of the shallowest goal node.

- Iterative deepening combines the benefits of depth-first and breadth-first-search
- Like depth-first-search,its memory requirements are modest;O(bd) to be precise.
- Like Breadth-first-search,it is complete when the branching factor is finite and optimal when the path cost is a non decreasing function of the depth of the node.
- The following figure shows the four iterations of ITERATIVE- DEEPENING_SEARCH on a binary search tree,where the solution is found on the fourth iteration.

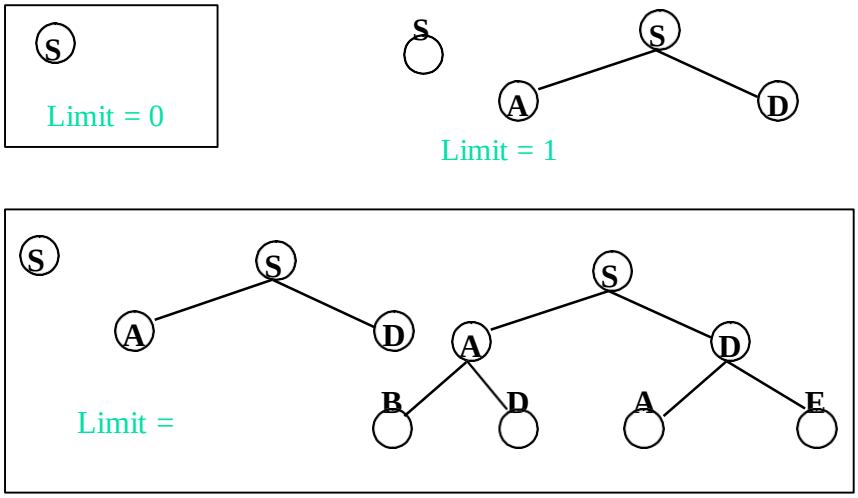
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
    end
```

The **iterative deepening search algorithm**, which repeatedly applies depth-limited-search with increasing limits. It terminates when a solution is found or if the depth limited search returns *failure*, meaning that no solution exists.




Four iterations of iterative deepening search on a binary tree

## Iterative deepening search




- Iterative search is not as wasteful as it might seem

**Properties of iterative deepening search**

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? No, unless step costs are constant
Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

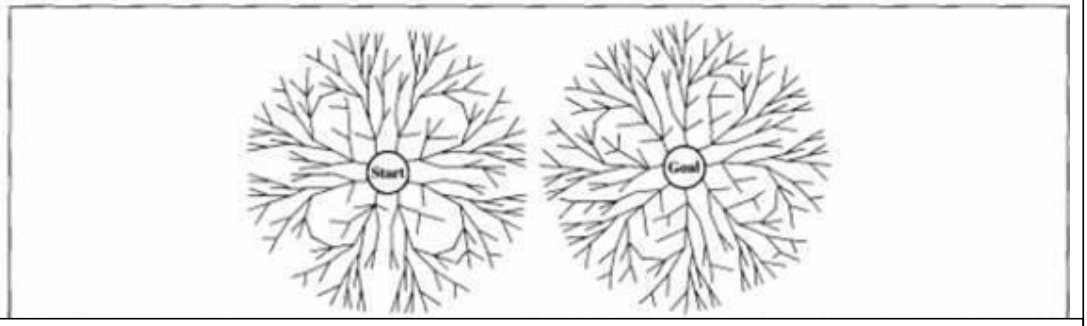
IDS does better because other nodes at depth $d$ are not expanded

BFS can be modified to apply goal test when a node is generated

- In general,iterative deepening is the prefered uninformed search method when there is a large search space and the depth of solution is not known.

## Bidirectional Search

- The idea behind bidirectional search is to run two simultaneous searches
  - ✓ one forward from the initial state and
  - ✓ other backward from the goal,
- It stops when the two searches meet in the middle.
- The motivation is that $b^{d/2} + b^{d/2}$ much less than $b^{d/}$




A schematic view of a bidirectional search that is about to succeed, when a Branch from the Start node meets a Branch from the goal node.

## Comparing Uninformed Search Strategies

The following table compares search strategies in terms of the four evaluation criteria.

| Criterion | Breadth-First | Uniform-Cost | Depth-First | Depth-Limited | Iterative Deepening | Bidirectional (if applicable) |
|---|---|---|---|---|---|---|
| Complete? | Yes[a] | Yes[a,b] | No | No | Yes[a] | Yes[a,d] |
| Time | $O(b^{d+1})$ | $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ | $O(b^m)$ | $O(b^\ell)$ | $O(b^d)$ | $O(b^{d/2})$ |
| Space | $O(b^{d+1})$ | $O(b^{1+\lfloor C^*/\epsilon \rfloor})$ | $O(bm)$ | $O(b\ell)$ | $O(bd)$ | $O(b^{d/2})$ |
| Optimal? | Yes[c] | Yes | No | No | Yes[c] | Yes[c,d] |

Evaluation of search strategies,b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs >= E for positive E; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

# INFORMED SEARCH AND EXPLORATION

## Informed (Heuristic) Search Strategies

- **Informed search strategy** is one that uses problem-specific knowledge beyond the definition of the problem itself.
- It can find solutions more efficiently than uninformed strategy.

## Best-first search

- **Best-first search** is an instance of general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function** f(n).
- The node with lowest evaluation is selected for expansion,because the evaluation measures the distance to the goal.
- This can be implemented using a priority-queue,a data structure that will maintain the fringe in ascending order of f-values.

## Heuristic functions

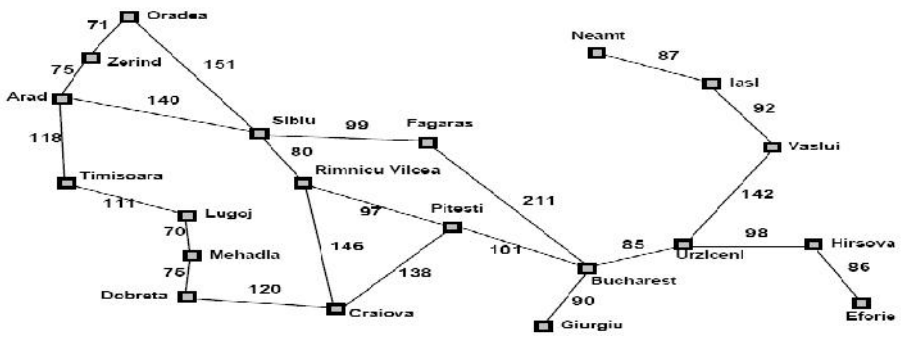
- A **heuristic function** or simply a **heuristic** is a function that ranks alternatives in various search algorithms at each branching step basing on an available information in order to make a decision which branch is to be followed during a search.
- The key component of Best-first search algorithm is a **heuristic function**,denoted by h(n):

  h(n) = estimated cost of the **cheapest path** from node n to a **goal node**.

- For example,in Romania, one might estimate the cost of the cheapest path from Arad to Bucharest via a **straight-line distance** from Arad to Bucharest
- Heuristic function are the most common form in which additional knowledge is imparted to the search algorithm.

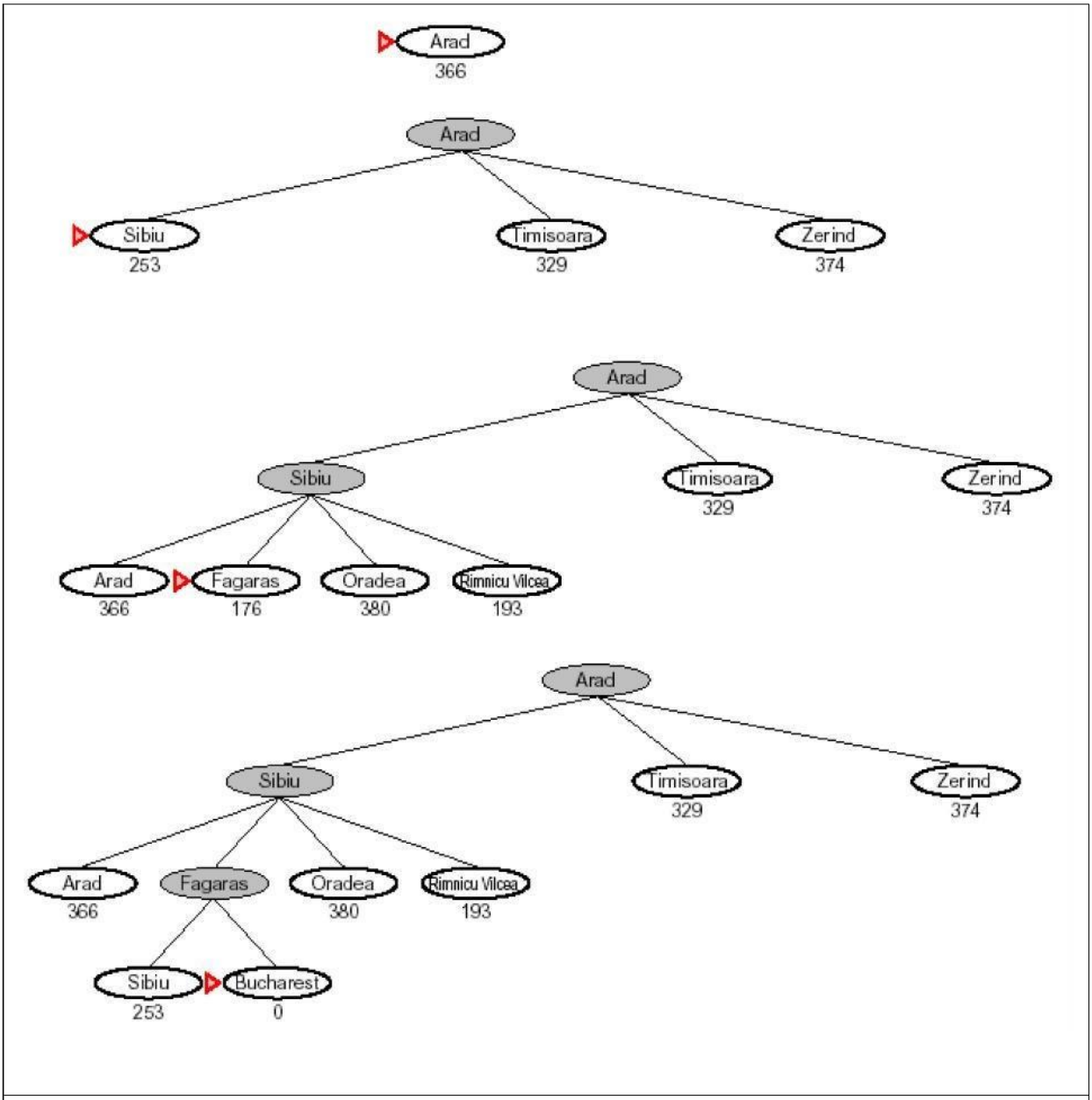
## Greedy Best-first search

- **Greedy best-first search** tries to expand the node that is closest to the goal,on the grounds that this is likely to a solution quickly.
- It evaluates the nodes by using the heuristic function f(n) = h(n).
- Taking the example of **Route-finding problems** in Romania, the goal is to reach Bucharest starting from the city Arad.
- We need to know the straight-line distances to Bucharest from various cities.
- For example, the initial state is In(Arad) ,and the straight line distance heuristic $h_{SLD}$(In(Arad)) is found to be 366.
- Vsing the **straight-line distance** heuristic $\mathbf{h_{SLD}}$ **,**the goal state can be reached faster.




| Straight line distance to Bucharest | |
|---|---|
| Arad | 366 |
| Bucharest | 0 |
| Craiova | 160 |
| Dobreta | 242 |
| Eforie | 161 |
| Fagaras | 178 |
| Giurgiu | 77 |
| Hirsova | 151 |
| Iasi | 226 |
| Lugoj | 244 |
| Mehadia | 241 |
| Neamt | 234 |
| Oradea | 380 |
| Pitesti | 98 |
| Rimnicu Vilcea | 193 |
| Sibiu | 253 |
| Timisoara | 329 |
| Urziceni | 80 |
| Vaslui | 199 |
| Zerind | 374 |

Values of $h_{SLD}$ - straight line distances to Bucharest




Strategies in greedy best-first search for Bucharest using straight-line distance heuristic $h_{SLD}$. Nodes are labeled with their h-values.

- The above figure shows the progress of greedy best-first search using $h_{SLD}$ to find a path from Arad to Bucharest.
- The first node to be expanded from Arad will be Sibiu,because it is closer to Bucharest than either Zerind or Timisoara.
- The next node to be expanded will be Fagaras,because it is closest.
- Fagaras in turn generates Bucharest,which is the goal.

### Properties of greedy search

- **Complete??** No-can get stuck in loops, e.g.,
  Iasi ! Neamt ! Iasi ! Neamt !
  Complete in finite space with repeated-state checking
- **Time??** O(bm), but a good heuristic can give dramatic improvement
- **Space??** O(bm)-keeps all nodes in memory
- **Optimal??** No

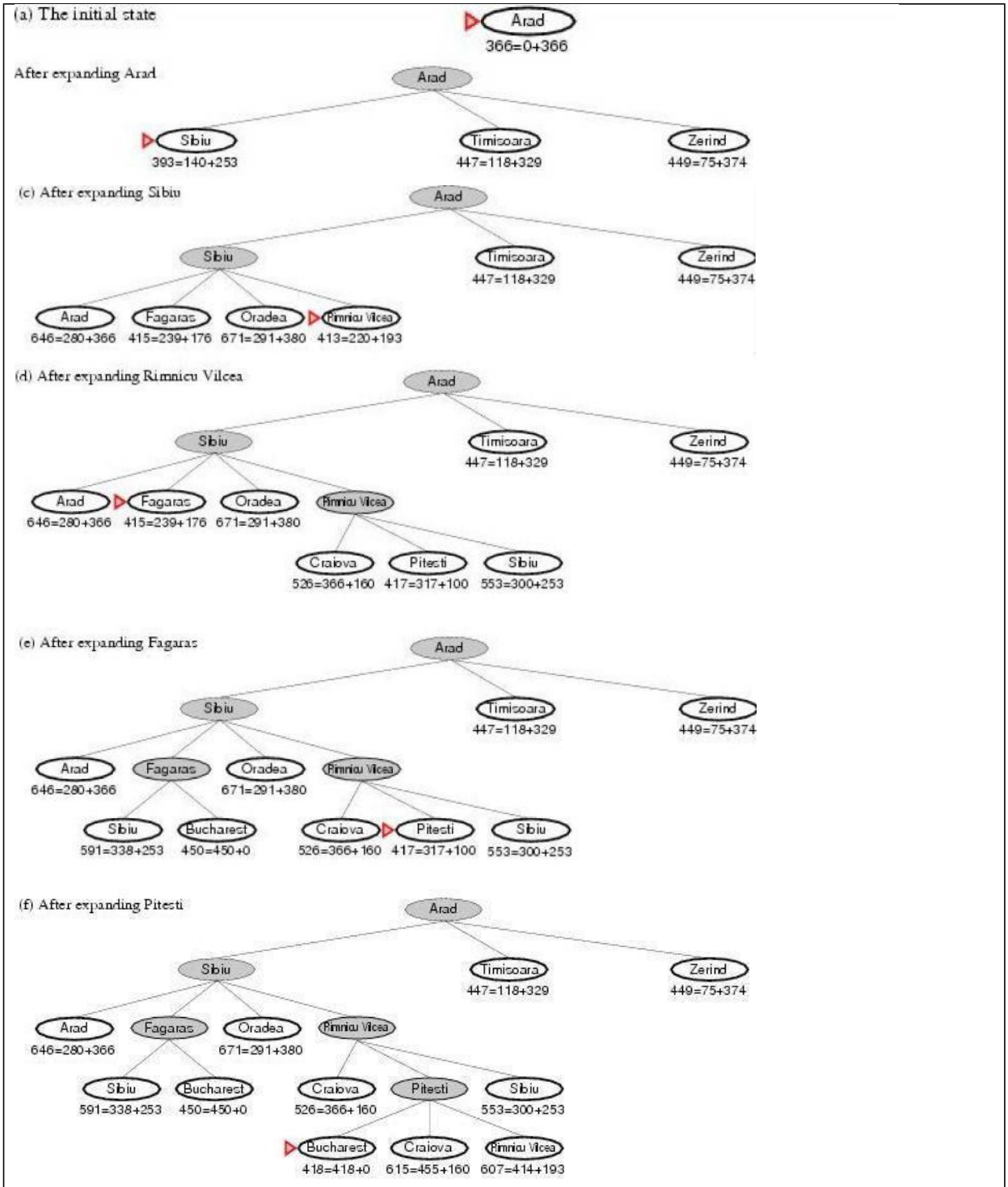
- Greedy best-first search is not optimal,and it is incomplete.
- The worst-case time and space complexity is $O(b^m)$,where m is the maximum depth of the search space.

## A* Search

- **A* Search** is the most widely used form of best-first search. The evaluation function f(n) is obtained by combining
  (1) **g(n) =** the cost to reach the node,and
  (2) **h(n) =** the cost to get from the node to the **goal** :
  $$f(n) = g(n) + h(n).$$
- $A^*$ Search is both optimal and complete. $A^*$ is optimal if h(n) is an admissible heuristic. The obvious example of admissible heuristic is the straight-line distance $h_{SLD}$.
- It cannot be an overestimate.
- $A^*$ Search is optimal if h(n) is an admissible heuristic - that is,provided that h(n) never overestimates the cost to reach the goal.
- An obvious example of an admissible heuristic is the straight-line distance $h_{SLD}$ that we used in getting to Bucharest.
- The progress of an $A^*$ tree search for Bucharest is shown in above figure.
- The values of 'g ' are computed from the step costs shown in the Romania,Also the values of $h_{SLD}$ are given in Figure Route Map of Romania.

## Recursive Best-first Search (RBFS)

- Recursive best-first search is a simple recursive algorithm that attempts to mimic the operation of standard best-first search,but using only linear space.
- The algorithm is shown in below figure.
- Its structure is similar to that of recursive depth-first search,but rather than continuing indefinitely down the current path,it keeps track of the f-value of the best alternative path available from any ancestor of the current node.
- If the current node exceeds this limit,the recursion unwinds back to the alternative path. As the recursion unwinds,RBFS replaces the f-value of each node along the path with the best f-value of its children.

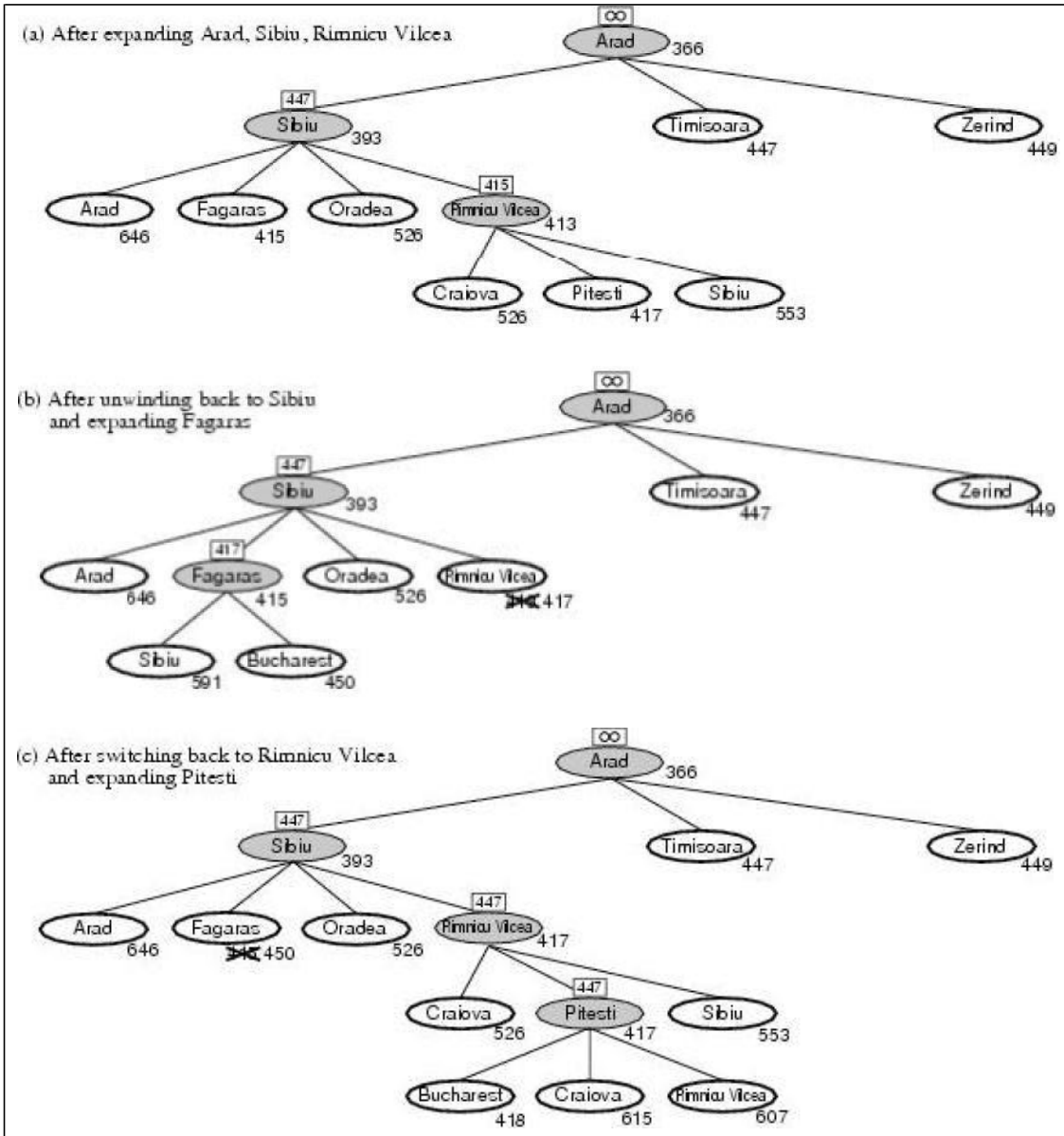



Stages in A* Search for Bucharest. Nodes are labeled with f = g + h . The h-values are the straight-line distances to Bucharest taken from figure Route map of Romania

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) return a solution or failure
    return  RFBS(problem,MAKE-NODE(INITIAL-STATE[problem]),∞)

function RFBS( problem, node, f_limit) return a solution or failure and a new f-
    cost limit
    if GOAL-TEST[problem](STATE[node]) then return node
    successors ← EXPAND(node, problem)
    if successors is empty then return failure, ∞
    for each s in successors do
        f [s] – max(g(s) + h(s), f [node])
    repeat
        best – the lowest f-value node in successors
        if f [best] > f_limit then return failure, f [best]
        alternative — the second lowest f-value among successors
        result, f [best] – RBFS(problem, best, min(f_limit, alternative))
        if result ↔ failure then return result
```

The algorithm for recursive best-first search



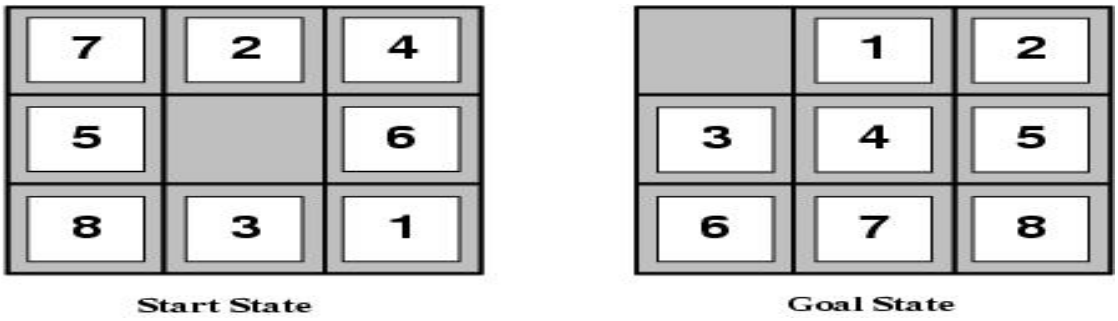

- Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node.
- (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras).
- (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea;then Fagaras is expanded,revealing a best leaf value of 450.
- (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed upto Fagaras; then Rimni Vicea is expanded.
- This time because the best alternative path(through Timisoara) costs atleast 447,the expansion continues to Bucharest

**RBFS Evaluation:**

- RBFS is a bit more efficient than IDA*
  - Still excessive node generation (mind changes)
- Like A*, optimal if *h(n)* is admissible
- Space complexity is *O(bd)*.
  - IDA* retains only one single number (the current f-cost limit)
- Time complexity difficult to characterize
  - Depends on accuracy if h(n) and how often best path changes.
- IDA* en RBFS suffer from ***too little*** memory.

# Heuristic Functions

- A **heuristic function** or simply a heuristic is a function that ranks alternatives in various search algorithms at each branching step basing on an available information in order to make a decision which branch is to be followed during a search




A typical instance of the 8-puzzle.

- The solution is 26 steps long.

**The 8-puzzle**

- The 8-puzzle is an example of Heuristic search problem.
- The object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration
- The average cost for a randomly generated 8-puzzle instance is about 22 steps.
- The branching factor is about 3.(When the empty tile is in the middle,there are four possible moves;when it is in the corner there are two;and when it is along an edge there are three).
- This means that an exhaustive search to depth 22 would look at about $3^{22}$ approximately = $3.1 \times 10^{10}$ states.
- By keeping track of repeated states, we could cut this down by a factor of about 170, 000, because there are only 9!/2 = 181,440 distinct states that are reachable.
- This is a manageable number, but the corresponding number for the 15-puzzle is roughly $10^{13}$.

- If we want to find the shortest solutions by using $A^*$,we need a heuristic function that never overestimates the number of steps to the goal.
- The two commonly used heuristic functions for the 15-puzzle are :

  (1) $h_1$ = the number of misplaced tiles.
- In the above figure all of the eight tiles are out of position, so the start state would have $h_1 = 8$. $h_1$ is an admissible heuristic.

  **(2)** $h_2$ = the sum of the distances of the tiles from their goal positions. This is called **the city block distance** or **Manhattan distance.**

$h_2$ is admissible ,because all any move can do is move one tile one step closer to the goal.

- Tiles 1 to 8 in start state give a Manhattan distance of

$$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18.$$

- Neither of these overestimates the true solution cost, which is 26.

**The Effective Branching factor**

- One way to characterize the **quality of a heuristic** is the **effective branching factor b*.** If the total number of nodes generated by A* for a particular problem is **N**,and the **solution depth** is **d,**then $b^*$ is the branching factor that a uniform tree of depth d would have to have in order to contain N+1 nodes. Thus,

$$N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$$

- For example,if $A^*$ finds a solution at depth 5 using 52 nodes,then effective branching factor is 1.92.
- A well designed heuristic would have a value of $b^*$ close to 1,allowing failru large problems to be solved.
- To test the heuristic functions $h_1$ and $h_2$,1200 random problems were generated with solution lengths from 2 to 24 and solved them with iterative deepening search and with $A^*$ search using both $h_1$ and $h_2$.
- The following table gives the average number of nodes expanded by each strategy and the effective branching factor.
- The results suggest that $h_2$ is better than $h_1$,and is far better than using iterative deepening search.
- For a solution length of 14,$A^*$ with $h_2$ is 30,000 times more efficient than uninformed iterative deepening search.

| | Search Cost | | | Effective Branching Factor | | |
|---|---|---|---|---|---|---|
| $d$ | IDS | $A^*(h_1)$ | $A^*(h_2)$ | IDS | $A^*(h_1)$ | $A^*(h_2)$ |
| 2 | 10 | 6 | 6 | 2.45 | 1.79 | 1.79 |
| 4 | 112 | 13 | 12 | 2.87 | 1.48 | 1.45 |
| 6 | 680 | 20 | 18 | 2.73 | 1.34 | 1.30 |
| 8 | 6384 | 39 | 25 | 2.80 | 1.33 | 1.24 |
| 10 | 47127 | 93 | 39 | 2.79 | 1.38 | 1.22 |
| 12 | 3644035 | 227 | 73 | 2.78 | 1.42 | 1.24 |
| 14 | – | 539 | 113 | – | 1.44 | 1.23 |
| 16 | – | 1301 | 211 | – | 1.45 | 1.25 |
| 18 | – | 3056 | 363 | – | 1.46 | 1.26 |
| 20 | – | 7276 | 676 | – | 1.47 | 1.27 |
| 22 | – | 18094 | 1219 | – | 1.48 | 1.28 |
| 24 | – | 39135 | 1641 | – | 1.48 | 1.26 |

Comparison of search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and $A^*$ Algorithms with $h_1$,and $h_2$. Data are average over 100 instances of the 8-puzzle,for various solution lengths.

# Inventing admissible heuristic functions

## Relaxed problems

- A problem with fewer restrictions on the actions is called a ***relaxed problem***
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- If the rules of the 8-puzzle are relaxed so that a tile can move *anywhere*, then *hi(n)* gives the shortest solution
- If the rules are relaxed so that a tile can move to *any adjacent square*, then *h2(n)* gives the shortest solution

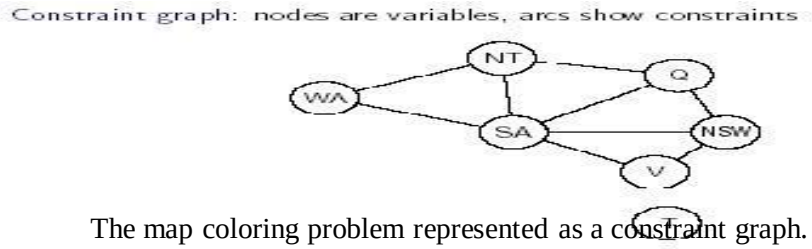
# CONSTRAINT SATISFACTION PROBLEMS (CSP)

- A **Constraint Satisfaction Problem** (or CSP) is defined by a
  - ✓ set of **variables** $X_1$, $X2….X_n$, and a
  - ✓ set of constraints $C_1,C_2,…,C_m$.
  - ✓ Each variable $X_i$ has a nonempty **domain** D,of possible **values**.
  - ✓ Each constraint $C_i$ involves some subset of variables and specifies the allowable combinations of values for that subset.
- A **State** of the problem is defined by an **assignment** of values to some or all of the variables,$\{X_i = v_i,X_j = v_j,…\}$.
- An assignment that does not violate any constraints is called a **consistent** or **legal assignment.**
- A complete assignment is one in which every variable is mentioned, and a **solution** to a CSP is a complete assignment that satisfies all the constraints.

- Some CSPs also require a solution that maximizes an **objective function**.
- For Example for Constraint Satisfaction Problem :
- The following figure shows the map of Australia showing each of its states and territories.
- We are given the task of coloring each region either red,green,or blue in such a way that the neighboring regions have the same color.
- To formulate this as CSP ,we define the variable to be the regions :WA,NT,Q,NSW,V,SA, and T.
- The domain of each variable is the set {red,green,blue}.
- The constraints require neighboring regions to have distinct colors;
- for example, the allowable combinations for WA and NT are the pairs {(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}.
- The constraint can also be represented more succinctly as the inequality WA not = NT,provided the constraint satisfaction algorithm has some way to evaluate such expressions.)
- There are many possible solutions such as
  { WA = red, NT = green,Q = red, NSW = green, V = red ,SA = blue,T = red}.


Principle states and territories of Australia. Coloring this map can be viewed as aconstraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color.

- It is helpful to visualize a CSP as a constraint graph,as shown in the following figure.
- The nodes of the graph corresponds to variables of the problem and the arcs correspond to constraints.




The map coloring problem represented as a constraint graph.

- CSP can be viewed as a standard search problem as follows :
  - **Initial state** : the empty assignment {},in which all variables are unassigned.
  - **Successor function** : a value can be assigned to any unassigned variable,provided that it does not conflict with previously assigned variables.
  - **Goal test** : the current assignment is complete.
  - **Path cost** : a constant cost(E.g.,1) for every step.

- Every solution must be a complete assignment and therefore appears at depth n if there are n variables.
- Depth first search algorithms are popular for CSPs

**Varieties of CSPs**

**(i) Discrete variables**

**Finite domains**

- The simplest kind of CSP involves variables that are **discrete** and have **finite domains.**
- Map coloring problems are of this kind.
- The 8-queens problem can also be viewed as finite-domain
- CSP,where the variables $Q_1$,$Q_2$,…..$Q_8$ are the positions each queen in columns 1,….8 and each variable has the domain {1,2,3,4,5,6,7,8}.
- If the maximum domain size of any variable in a CSP is d,then the number of possible complete assignments is $O(d^n)$ - that is,exponential in the number of variables.
- Finite domain CSPs include **Boolean CSPs**,whose variables can be either *true* or *false.*

**Infinite domains**

- Discrete variables can also have **infinite domains** - for example,the set of integers or the set of strings.
- With infinite domains,it is no longer possible to describe constraints by enumerating all allowed combination of values. Instead a constraint language of algebric inequalities such as $Startjob_1 + 5 <= Startjob_3$.

**(ii) CSPs with continuous domains**

- CSPs with continuous domains are very common in real world.
- For example ,in operation research field,the scheduling of experiments on the Hubble Telescope requires very precise timing of observations; the start and finish of each observation and maneuver are continuous-valued variables that must obey a variety of astronomical,precedence and power constraints.
- The best known category of continuous-domain CSPs is that of **linear programming** problems,where the constraints must be linear inequalities forming a *convex* region.
- Linear programming problems can be solved in time polynomial in the number of variables.

**Varieties of constraints :**

**(i) Unary constraints** involve a single variable.

Example: SA # green

(ii) **Binary constraints** involve pairs of variables.

Example: SA # WA

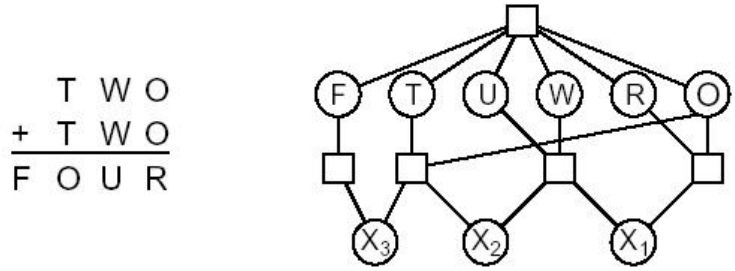
(iii) **Higher order constraints** involve 3 or more variables.

Example: cryptarithmetic puzzles.

(iv) **Absolute constraints** are the constraints, which rules out a potential solution when they are violated

(v) **Preference constraints** are the constraints indicating which solutions are preferred

Example: Vniversity Time Tabling Problem




Variables: $F\ T\ U\ W\ R\ O\ X_1\ X_2\ X_3$
Domains: $\{0,1,2,3,4,5,6,7,8,9\}$
Constraints
$alldiff(F,T,U,W,R,O)$
$O + O = R + 10 \cdot X_1$, etc.

- Cryptarithmetic problem.
- Each letter stands for a distinct digit;the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct,with the added restriction that no leading zeros are allowed.
- The constraint hypergraph for the cryptarithmetic problem,showint the *Alldiff* constraint as well as the column addition constraints.
- Each constraint is a square box connected to the variables it contains.

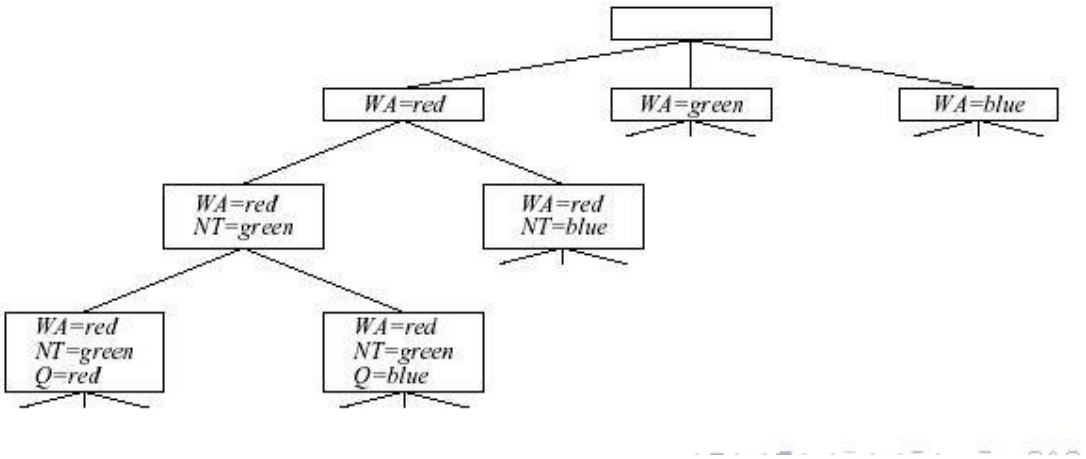
## Backtracking Search for CSPs

- The term **backtracking search** is used for depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign.
- The following algorithm shows the Backtracking Search for CSP

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
  return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment given CONSTRAINTS[csp] then
          add {var = value} to assignment
          result ← RECURSIVE-BACKTRACKING(assignment, csp)
          if result ≠ failure then return result
          remove {var = value} from assignment
  return failure
```

**A simple backtracking algorithm for constraint satisfaction problem. The algorithm is modeled on the recursive depth-first search**




**Part of search tree generated by simple backtracking for the map coloring problem.**

## Propagating information through constraints

- So far our search algorithm considers the constraints on a variable only at the time that the Variable is chosen by SELECT-VNASSIGNED-VARIABLE.
- But by looking at some of the constraints earlier in the search, or even before the search has started, we can drastically reduce the search space.

## Forward checking

- One way to make better use of constraints during search is called **forward checking**.
- Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y 's domain any value that is inconsistent with the value chosen for X.
- The following figure shows the progress of a map-coloring search with forward checking.

| | *WA* | *NT* | *Q* | *NSW* | *V* | *SA* | *T* |
|---|---|---|---|---|---|---|---|
| Initial domains | R G B | R G B | R G B | R G B | R G B | R G B | R G B |
| After *WA*=*red* | (R) | G B | R G B | R G B | R G B | G B | R G B |
| After *Q*=*green* | (R) | B | (G) | R B | R G B | B | R G B |
| After *V*=*blue* | (R) | B | (G) | R | (B) | | R G B |

**Figure 5.6** The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes *red* from the domains of the neighboring variables $NT$ and $SA$. After $Q = green$, *green* is deleted from the domains of $NT$, $SA$, and $NSW$. After $V = blue$, *blue* is deleted from the domains of $NSW$ and $SA$, leaving $SA$ with no legal values.

## Constraint propagation

- Although forward checking detects many inconsistencies, it does not detect all of them.
- **Constraint propagation** is the general term for propagating the implications of a constraint on one variable onto other variables.

## Arc Consistency

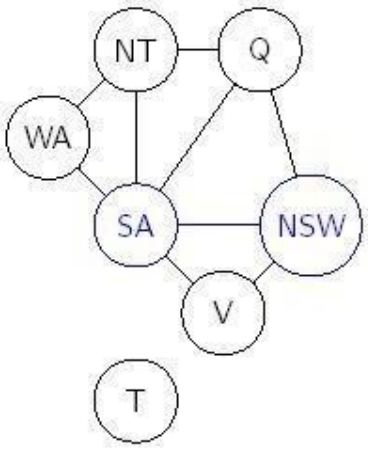



Figure: Australian Territories

- One method of constraint propagation is to enforce **arc consistency**
  - Stronger than forward checking
  - Fast
- *Arc* refers to a *directed* arc in the constraint graph
- Consider two nodes in the constraint graph (e.g., *SA* and *NSW*)
  - An arc is **consistent** if
  - For every value $x$ of *SA*
  - There is some value $y$ of *NSW* that is consistent with $x$
- Examine arcs for consistency in *both* directions

## K-Consistency

- Can define stronger forms of consistency

**$k$-Consistency**

A CSP is **$k$-consistent** if, for any consistent assignment to $k-1$ variables, there is a consistent assignment for the $k$-th variable

- **1-consistency (node consistency)**
  - Each variable by itself is consistent (has a non-empty domain)
- **2-consistency (arc consistency)**
- **3-consistency (path consistency)**
  - Any pair of adjacent variables can be extended to a third

## Local Search for CSPs

- Local search algorithms good for many CSPs
- Use complete-state formulation
  - Value assigned to every variable
  - Successor function changes one value at a time
- Have already seen this
  - Hill climbing for 8-queens problem (AIMA § 4.3)
- Choose values using **min-conflicts** heuristic
  - Value that results in the minimum number of conflicts with other variables

## The Structure of Problems

## Problem Structure

- Consider ways in which the structure of the problem's constraint graph can help find solutions
- Real-world problems require decomposition into subproblems

## Independent Sub problems

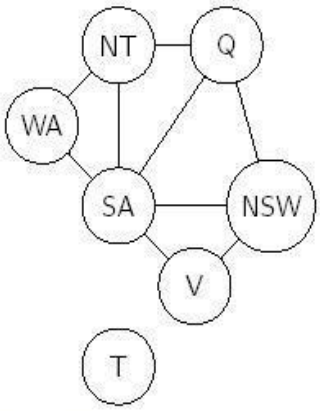



Figure: Australian Territories

- $T$ is not connected
- Coloring $T$ and coloring remaining nodes are **independent subproblems**
- *Any* solution for $T$ combined with *any* solution for remaining nodes solves the problem
- Independent subproblems correspond to **connected components** of the constraint graph
- Sadly, such problems are rare

## Tree-Structured CSPs

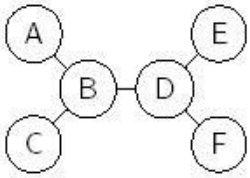



Figure: Tree-Structured CSP

- In most cases, CSPs are connected
- A simple case is when the constraint graph is a **tree**
- Can be solved in time linear in the number of variables
  - Order variables so that each parent precedes its children
  - Working "backward," apply arc consistency between child and parent
  - Working "forward," assign values consistent with parent

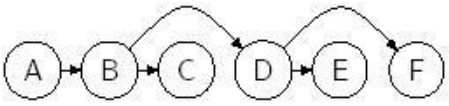



Figure: Linear ordering

## UNIT-II REASONING WITH LOWER ORDER LOGICS

## 2.0 Logic:

- A knowledge representation language in which syntax and semantics are defined correctly is known as logic.
- A formal language to represent the knowledge in which reasoning is carried out to achieve the goal state.
- Logics consists of **the** following two representations in sequence,
  - A formal system is used to describe the state of the world
    - ✓ Syntax " Which describes how to make sentences"
    - ✓ Semantics " Which describes the meaning of the sentences"
  - The proof theory "a set of rule for deducing the entailments of a set of sentences.
- We will represent the sentences using two different logics, They are,
  - Propositional Logic (or) Boolean logic
  - Predicate logic (or) First order logic

## 2.1 Logical Agents:

- The Logical agent has to perform the following task using logic representation. The tasks are,
  - ✓ To know the current state of the world
  - ✓ How to infer the unseen properties of the world
  - ✓ New changes in the environment
  - ✓ Goal of the agent

How to perform actions depends on circumstances?

## 2.2 Propositional Logic:

- Each fact is represented by one symbol.
- Proposition symbols can be connected with Boolean connectives, to give more complex meaning. Connectives ,
  - **Λ** Logical Conjunction
  - **V** Logical disjunction
  - ¬ Negation
  - ⇔ Material Equivalence or Conditional
  - ⇒ Material Implication or conditional
- Simple statements are implemented
- The Symbols of propositional logic are the logical constants, (True and False)
- For Example: - P, Q

**Connectives**

**Λ (and)----------Example: - P Λ Q**

**V (or)----------Example: - P V Q**
**⇒ (implies)----------Example: - (P Λ Q) ⇒ R**
**⇔ (equivalent)----------Example: - (P Λ Q) ⇔ (Q Λ P)**
**¬ (not)----------Example: - ¬ P**

- A **BNF** (Backus-Naur Form) grammar of sentence in propositional logic

**Sentence----------> Atomic sentence | complex sentence**
**Atomic sentence----------> True | False | P | Q | R |**
**Complex sentence ---------> (sentence) | sentence connective sentence | ¬ Sentence**
**Connective----------> Λ | V | ⇒ | ⇔**

- Order of precedence ( from highest to lowest) : **¬ , Λ , V , ⇒ and ⇔**
- Example : - ¬P VQ ΛR ⇒ S is equivalent to ((¬P )V(Q Λ R)) ⇒ S
- The following truth table shows the five logical connectives

| P | Q | ¬ P | P Λ Q | P V Q | P ⇒ Q | P ⇔ Q |
|---|---|---|---|---|---|---|
| False | False | True | False | False | True | True |
| False | True | True | False | True | True | False |
| True | False | False | False | True | False | False |
| True | True | False | True | True | True | True |

- These truth table can be used to define the validity of a sentence.
- If the sentence is true in every row (i.e. for different types of logical constants) then the sentence is a valid sentence.
- For Example: - ((P V H) Λ¬H) ⇒ P Check whether the given sentence is a valid sentence or not.

| P | H | P V H | (P V H) Λ¬H | ((P V H) Λ¬H) ⇒ P |
|---|---|---|---|---|
| False | False | False | False | True |
| False | True | True | False | True |
| True | False | True | True | True |
| True | True | True | False | True |

- The given sentence is ((P V H) Λ¬H) ⇒ P valid sentence, because the sentence is TRUE in every row for different types of logical statements.

## 2.3 Inference Rules for Propositional logic:

- The propositional logic has seven inference rules.
- Inference means conclusion reached by reasoning from data or premises; speculation.
- A procedure which combines known facts to produce ("infer") new facts.
- **Logical inference** is used to create new sentences that logically follow from a given set of predicate calculus sentences (KB).
- An inference rule is **sound** if every sentence X produced by an inference rule operating on a KB logically follows from the KB. (That is, the inference rule does not create any contradictions)
- An inference rule is **complete** if it is able to produce every expression that logically follows from (is entailed by) the KB. (Note the analogy to complete search algorithms.)
- Here are some examples of sound rules of inference

  A rule is sound if its conclusion is true whenever the premise is true

| Rule | Premise | Conclusion |
|---|---|---|
| Modus Ponens | A, A → B | B |
| And Introduction | A, B | A ∧ B |
| And Elimination | A ∧ B | A ∨ B |
| Or Introduction | A, B | A |
| Double Negation | ¬¬A | A |
| Unit Resolution | A ∨ B, ¬B | A |
| Resolution | A ∨ B, ¬B ∨ C | A ∨ C |

## 2.4 An Agent for the Wumpus world – Propositional logic

- We will discuss the knowledge base representation and a method to find the wumpus using propositional logic representation.
- From the following figure assume that the agent has reached the square (1,2)

- **The Knowledge Base:** The agent percepts are converted into sentences and entered into the knowledge base, with some valid sentences that are entailed by the percept sentences
- From the above figure we can perceive the following percept sentences and it is added to the knowledge base.

$\neg S_{1,1}$ $\neg B_{1,1}$ ---------- for the square (1, 1)
$\neg S_{2,1}$ $\neg B_{2,1}$ ---------- for the square (2, 1)
$S_{1,2}$ $\neg B_{1,2}$ ---------- for the square (1, 2)
$S_{1,2}$ ---------- There is a stench in (1, 2)
$\neg B_{1,2}$ ---------- There is a breeze in (1, 2)
$\neg S_{2,1}$ ---------- There is a stench in (2, 1)
$B_{2,1}$ ---------- There is a breeze in (2, 1)
$\neg S_{1,1}$ ---------- There is a stench in (1, 1)
$\neg B_{1,1}$ ---------- There is a breeze in (1, 1)

- The rule of three squares,
  - $R_{1}$ :- $\neg S_{1,1} \Rightarrow \neg W_{1\,1} \wedge \neg W_{1\,2} \wedge \neg W_{2\,1}$
  - $R_{2}$:- $\neg S_{2,1} \Rightarrow \neg W_{1\,1} \wedge \neg W_{2\,2} \wedge \neg W_{2\,1} \wedge \neg W_{3\,1}$
  - $R_{3}$:- $\neg S_{1,2} \Rightarrow \neg W_{1\,1} \wedge \neg W_{1\,2} \wedge \neg W_{2\,2} \wedge \neg W_{1\,3}$
  - $R_{4}$:- $\neg S_{1,2} \Rightarrow W_{1\,3} \vee W_{1\,2} \vee W_{2\,2} \vee W_{1\,1}$

- Finding the wumpus as, We can prove that the Wumpus is in (1, 3) using the four rules given.
  - Apply Modus Ponens with $\neg S_{1\,1}$ and $R_1$:
    - $\neg W_{1\,1} \wedge \neg W_{1\,2} \wedge \neg W_{2\,1}$
  - Apply And-Elimination to this, yielding three sentences:
    - $\neg W_{1\,1}, \neg W_{1\,2}, \neg W_{2\,1}$
  - Apply Modus Ponens to $\neg$S21 and R2, then apply And-elimination:
    - $\neg W_{2\,2}, \neg W_{2\,1}, \neg W_{3\,1}$
  - Apply Modus Ponens to S12 and R4 to obtain:
    - $W_{1\,3} \vee W_{1\,2} \vee W_{2\,2} \vee W_{1\,1}$
  - Apply Unit resolution on $(W_{1\,3} \vee W_{1\,2} \vee W_{2\,2} \vee W_{1\,1})$ and $\neg W_{1\,1}$:
    - $W_{1\,3} \vee W_{1\,2} \vee W_{2\,2}$
  - Apply Unit Resolution with $(W_{1\,3} \vee W_{1\,2} \vee W_{2\,2})$ and $\neg W_{2\,2}$:
    - $W_{1\,3} \vee W_{1\,2}$
  - Apply Unit Resolution with $(W_{1\,3} \vee W_{1\,2})$ and $\neg W_{1\,2}$:
    - $W_{1\,3}$

## 2.5 First-Order Logic:

- **First-Order Logic** is a logic which is sufficiently expressive to represent a good deal of our commonsense knowledge.
- It is also either includes or forms the foundation of many other representation languages.
- It is also called as **First-Order Predicate calculus.**
- It is abbreviated as **FOL** or **FOPC**

### 2.5.1 Representation Revisited:

- It is necessary to know about the nature of representation languages.
- The following are the some languages,
    - ✓ Programming languages
    - ✓ Propositional logic languages
    - ✓ Natural languages

## Programming languages:

- Programming languages like C++ or Java are the largest class of formal languages in common use.
- Programs represent only computational processes.
- Data structures within programs can represent facts.
- For Example, 4 x 4 arrays can be used by a program to represent the contents of the Wumpus world.
- Thus the programming language statement *World [2, 2]* ⟵ *Pit* is a fairly natural way to assert that there is a pit in square [2, 2].

**Disadvantages:**

- Programming languages lack is any general mechanism for deriving facts from other facts; each update to a data structure is done by a domain-specific procedure whose details are derived by the programmer from his or her own knowledge of the domain.
- A second drawback of data structures in programs is the lack of any easy way to say
- For Example, "There is a Pit in [2,2] or [3,1]" or "If the Wumpus is in [I,I] then he is not in [2,2]".
- Programs lack the expressiveness required to handle partial information.

## Propositional Logic Languages:

- Propositional logic is a declarative language
- The following are the properties of propositional logic
- Its semantics is based on a truth relation between sentences and possible worlds.
- It also has sufficient expressive power to deal with partial information, using disjunction and negation.

- It also has **compositionality** that is desirable in representation languages namely **compositionality.**
- In a **compositionality** language, the meaning of a sentence is function of the meaning of its parts.
- For Example, "$S_{1,4} \wedge S_{1,2}$" is related to the meanings of "$S_{1,4}$" and "$S_{1,2}$"
- It would be very strange if "$S_{1,4}$" meant that there is a stench in square [1,4] and "$S_{1,2}$" meant that there is a stench in square [1,2], but "$S_{1,4} \wedge S_{1,2}$" meant that France and Poland drew 1-1 in last week's ice hockey qualifying match.
- Clearly, non- Compositionality makes life much more difficult for the reasoning system.

**Advantages:**

- Declarative
- Context-Independent
- Unambiguous

**Disadvantages:**

- It lacks the expressive power to describe an environment with many objects concisely.
- For Example, it is forced to write a separate rule about breezes and pits for each square, such as $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$.
- The procedural approach of programming languages can be contrasted with the declarative nature of propositional logic, in which knowledge and inference are separate and inference is entirely domain-independent.

## Natural Languages:

- A moments thought suggests that natural languages like English are very expressive indeed.
- Natural language is essentially a declarative knowledge representation language and attempts to pin down its formal semantics.
- The modern view of natural language is that it serves a somewhat different purpose, namely as a medium for communication rather than pure representation.
- When a speaker points and says, "Look!" the listener comes to know that, say, Superman has finally appeared over the rooftops.
- The meaning of the above sentence depends both on the sentence itself and on the context in which the sentence was spoken.

**Disadvantages:**

- It is difficult to understand how the context can be represented.
- This is because one could not store a sentence such as "Look!" in knowledge base and expect to recover its meaning without also storing a representation of the context.
- They are also non-compositional
- They suffer from ambiguity, which would cause difficulties for thinking.
- For Example, when people think about spring, they are not confused as to whether they are thinking about a season or something that goes *boing-and* if one word can correspond to two thoughts, thoughts can't be words.

### 2.5.2 First-Order Logic:

- **First-Order Logic** is a logic which is sufficiently expressive to represent a good deal of our commonsense knowledge.
- It is also either includes or forms the foundation of many other representation languages.
- It is also called as **First-Order Predicate calculus.**
- It is abbreviated as **FOL** or **FOPC**
- **FOL** adopts the foundation of propositional logic with all its advantages to build a more expressive logic on that foundation, borrowing representational ideas from natural language while avoiding its drawbacks
- The Syntax of natural language contains elements such as,

  Nouns and noun phrases that refer to objects (Squares, pits, rumpuses)

  Verbs and verb phrases that refer to among objects ( is breezy, is adjacent to)
- Some of these relations are functions-relations in which there is only one "Value" for a given "input".
- For Example,

  **Objects:** People, houses, numbers

  **Relations:** These can be unary relations or properties such as red, round,

  More generally n-ary relations such as brother of, bigger than,

  **Functions:** father of, best friend,…
- Indeed, almost any assertion can be thought of as referring to objects and properties or relations
- For Example, in the way of Sentence " One plus Two is Three"
- Where,

**Objects:** One, Two, Three, One plus Two

**Relations:** equals

**Function:** plus

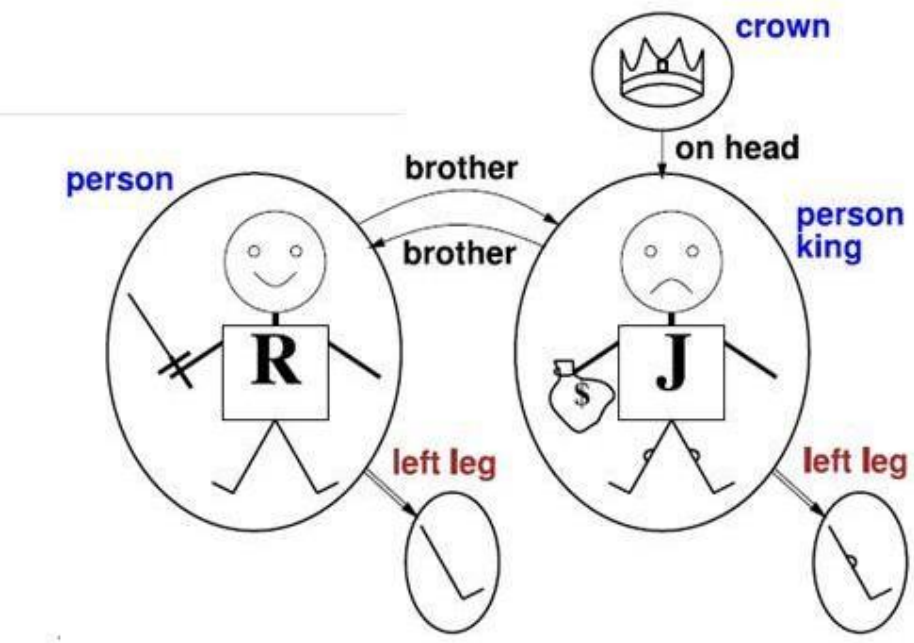
- Ontological commitment of First-Order logic language is “Facts”, “Objects”, and “Relations”.
- Where ontological commitment means “WHAT EXISTS IN THE WORLD”.
- Epistemological Commitment of First-Order logic language is “True”, “False”, and “Unknown”.
- Where epistemological commitment means “WHAT AN AGENT BELIEVES ABOUT FACTS”.

**Advantages:**

- It has been so important to mathematics, philosophy, and Artificial Intelligence precisely because those fields can be usefully thought of as dealing with objects and the relations among them.
- It can also express facts about some or all of the objects in the universe.
- It enables one to represent general laws or rules, such as the statement “Squares neighboring the wumpus are smelly”.

### 2.5.3 Syntax and Semantics of First-Order Logic:

- The models of a logical language are the formal structures that constitute the possible worlds under consideration.
- Models for propositional logic are just sets of truth values for the proposition symbols.
- Models for first-order logic are more interesting.
- First they have objects in them.
- The domain of a model is the set of objects it contains; these objects are sometimes called domain elements.
- The following diagram shows a model with five objects




- The five objects are,
    - ✓ Richard the Lionheart
    - ✓ His younger brother
    - ✓ The evil King John
    - ✓ The left legs of Richard and John
    - ✓ A crown
- The objects in the model may be related in various ways, In the figure Richard and John are brothers.
- Formally speaking, a relation is just the set of tuples of objects that are related.
- A tuple is a collection of Objects arranged in a fixed order and is written with angle brackets surrounding the objects.
- Thus, the brotherhood relation in this model is the set

  **{(Richard the Lionheart, King John),(King John, Richard the Lionheart)}**
- The crown is on King John's head, so the "on head" relation contains just one tuple, (the crown, King John).
- The relation can be binary relation relating pairs of objects (Ex:- "Brother") or unary relation representing a common object (Ex:- "Person" representing both Richard and John)

- Certain kinds of relationships are best considered as functions that relates an object to exactly one object.
- For Example:- each person has one left leg, so the model has a unary "left leg" function that includes the following mappings

  (Richard the Lionheart)-----> Richard's left leg

  (King John) ----> John's left leg

➢ **Symbols and Interpretations:**

- The basic syntactic elements of first-order logic are the symbols that stand for **objects, relations** and **functions**

❖ **Kinds of Symbols**

- The symbols come in three kinds namely,
  - ✓ Constant Symbols standing for **Objects** (Ex:- Richard)
  - ✓ Predicate Symbols standing for **Relations** (Ex:- King)
  - ✓ Function Symbols stands for **functions** (Ex:- LeftLeg)
    - o Symbols will begin with uppercase letters
    - o The choice of names is entirely up to the user
    - o Each predicate and function symbol comes with an arity
    - o Arity fixes the number of arguments.
- The semantics must relate sentences to models in order to determine truth.
- To do this, an interpretation is needed specifying exactly which **objects, relations** and **functions** are referred to by the **constant, predicate and function symbols**.
- One possible interpretation called as the intended interpretation- is as follows;
  - ✓ **Richard** refers to **Richard the Lionheart** and **John** refers to the **evil King John.**
  - ✓ **Brother** refers to the brotherhood relation, that is the set of tuples of objects given in equation **{(Richard the Lionheart, King John),(King John, Richard the Lionheart)}**
  - ✓ **OnHead** refers to the "on head" relation that holds between the crown and King John; **Person, King** and **Crown** refer to the set of objects that are persons, kings and crowns.
  - ✓ **Leftleg** refers to the "left leg" function, that is, the mapping given in **{(Richard the Lionheart, King John),(King John, Richard the Lionheart)}**

- There are many other possible interpretations relating these symbols to this particular model.
- The truth of any sentence is determined by a model and an interpretation for the sentence symbols.
- The syntax of the FOL with equality specified in BNF is as follows

| | | |
|---|---|---|
| Sentence | ⟶ | AtomicSentence |
| | \| | (Sentence Connective Sentence) |
| | \| | Quantifier Variable,…Sentence |
| | \| | - Sentence |
| AtomicSentence | ⟶ | Predicate (Term…) \| Term = Term |
| Term | ⟶ | Function (Term,….) |
| | \| | Constatnt |
| | \| | Variable |
| Connective | ⟶ | ⇒ \| **Λ** \| **V** \| ⇔ |
| Quantifier | ⟶ | ∀ \| ∃ |
| Constant | ⟶ | **A \| $X_1$ \| John** |
| Variable | ⟶ | **a \| x \| s \| …** |
| Predicate | ⟶ | **Before \| HasColor \| Raining \| ….** |
| Function | ⟶ | **Mother \| LeftLeg \| ….** |

- Where,

| | |
|---|---|
| **Λ** | Logical Conjunction |
| **V** | Logical disjunction |
| ∀ | Universal Quantification |
| ∃ | Existential Quantification |
| ⇔ | Material Equivalence |
| ⇒ | Material Implication |

❖ **Terms:**

- A Term is a logical expression that refers to an object
- Constant symbol are therefore terms, but it is not always convenient to have a distinct symbol to name every object.
- For Example:- in English we might use the expression “King Johns left leg” rather than giving a name to his leg.

- A complex term is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol.
- It is just like a complicated kind of name. It's not a "subroutine call" that returns a value".
- The formal semantics of terms is straight forward,

  Consider a term $f(t1 \ldots\ldots t_n)$

  Where

  f- some function in the model (call it F)

  The argument terms – objects in the domain

  The term – object that is the value of the function F applied to the domain

- For Example:- suppose the **LeftLeg** function symbol refers to the function is,

  (Richard the Lionheart) ------- Richards left leg

  (King John )-------- Johns left leg

  **John** refers to King John, then **LeftLeg** (John) refers to king Johns left leg.

- In this way the Interpretation fixes the referent of every term.

❖ **Atomic Sentences:-**

- An atomic sentence is formed from a predicate symbol followed by a parenthesized list of terms:

  **Brother (Richard, John)**

- This states that Richard the Lionheart is the brother of King John.
- Atomic Sentences can have complex terms as arguments.
- Thus, **Married(Father(Richard), Mother(John))** states that Richard the Lionheart's father is married to King John's mother
- An atomic sentence is true in a given model, under a given interpretation, if the relation referred to by the predicate symbol holds among the objects referred to by the arguments.

❖ **Complex Sentences:-**

- Logical connectives can be used to construct more complex sentences, just as in propositional calculus.
- The semantics of sentences formed with logical connectives is identical to that in the propositional case.

¬ Brother (LeftLeg (Richard), John)

Brother (Richard, John) Λ Brother (John, Richard)

King (Richard) V King (John)

¬ King (Richard) ⇒ King (John)

¬ **it refers " Logical Negation"**

❖ **Quantifiers:-**

- Quantifiers are used to express properties of entire collections of objects, instead of enumerating the objects by name if a logic that allows object is found.
- It has two type,
- The following are the types of standard quantifiers,

  ✓ Universal
  ✓ Existential

❖ **Universal Quantification (∀):-**

- Universal Quantification make statement about every object.
- "All Kings are persons", is written in first-order logic as

  $\forall_x$ **king (x) ⇒ Person (x)**

- ∀ is usually pronounced "For all….", Thus the sentences says , "For all x, if x is a king, then x is a person".
- The symbol x is called a variable.
- A variable is a term all by itself, and as such can also serve as the argument of a function-for example, **LeftLeg(x)**.
- A term with no variables is called a **ground term.**
- Based on our model, we can extend the interpretation in five ways,

  x----------Richard the Lionheart

  x----------King John

  x----------Richard's Left leg

  x----------John's Left leg

  x----------the crown

- The universally quantified sentence is equivalent to asserting the following five sentences

  Richard the Lionheart ------ Richard the Lionheart is a person

King John is a King---------- King John is a Person

Richard's left leg is King ----------Richard's left leg is a person

John's left leg is a King----------John's left leg is a person

The crown is a King ---------- The crown is a Person

❖ **Existential Quantification (∃):-**

- An existential quantifier is used make a statement about some object in the universe without naming it.
- To say, for example :- that King John has a crown on his head, write ∃x crown (x) Λ OnHead (x, John).
- ∃x is pronounced " There exists an x such that..," or "For some x.."
- Consider the following sentence,

$$\exists_x \textbf{ crown (x)} \Rightarrow \textbf{ OnHead (x,John)}$$

- Applying the semantics says that at least one of the following assertion is true,

| | |
|---|---|
| Richard the Lionheart is a crown | Λ Richard the Lionheart is on John's head |
| King John is Crown | Λ King John is on John's head |
| Richard's left leg is a crown | Λ Richard's left leg is on John's head |
| John's left leg is a crown | Λ John's left leg is on John's head |
| The crown is a crown | Λ The crown is on John's head |

- Now an implication is true if both premise and conclusion are true, or if its premise is false.

❖ **Nested Quantifiers:-**

- More complex sentences are expressed using multiple quantifiers.
- The following are the some cases of multiple quantifiers,
- The simplest case where the quantifiers are of the same type.
- For Example:- "Brothers are Siblings" can be written as

$$\forall_x \forall_{y,} \textbf{Brother (x,y)} \Rightarrow \textbf{sibling (x,y)}$$

- Consecutive quantifiers of the same type can be written as one quantifier with several variables.
- For Example:- to say that siblinghood is a symmetric relationship as

$$\forall_{x,y} \textbf{sibling (x,y)} \Leftrightarrow \textbf{sibling (y,z)}$$

- In some cases it is possible to have mixture of quantifiers.
- For Example:- "Everybody loves somebody" means that for every person, there is someone that person loves:

$\forall_x \exists_y$ Loves (x, y).

- On the other hand , to say "There is someone who is loved by everyone", we can write as

$\exists_y \forall_x$ Loves (x, y).

❖ **Connections between $\forall$ and $\exists$**

- The two quantifiers are actually intimately connected with each other , through negation,
- Declaring that everyone dislikes parsnips is the same as declaring there does not exist someone who likes them, and vice versa:

$\forall_x \neg$ Likes (x, Parsnips) is equivalent to $\neg \exists_x$ Likes (x, Parsnips)

- Going one step further: "Everyone likes ice cream" means that there os no one who does not like ice cream:

$\forall_x \neg$ Likes (x, IceCream) is equivalent to $\neg \exists_x$ Likes (x, IceCream)

- Because $\forall$ is really a conjunction over the universe of objects $\exists$ is a disjunction, it should not be surprising that they obey de Morgan's rules.
- The de Morgan's rules for quantified and un-quantified sentences are as follows:
- **≡ it refers definition**

| | |
|---|---|
| $\forall_x \neg \mathbf{P} \equiv \neg \exists_x \mathbf{P}$ | $\neg \mathbf{P} \wedge \neg \mathbf{Q} \equiv \neg (\mathbf{P} \vee \mathbf{Q})$ |
| $\neg \forall_x \mathbf{P} \equiv \exists_x \neg \mathbf{P}$ | $\neg (\mathbf{P} \wedge \mathbf{Q}) \equiv \neg \mathbf{P} \vee \neg \mathbf{Q}$ |
| $\forall_x \mathbf{P} \equiv \neg \exists_x \neg \mathbf{P}$ | $\mathbf{P} \wedge \mathbf{Q} \equiv \neg (\neg \mathbf{P} \vee \neg \mathbf{Q})$ |
| $\exists_x \mathbf{P} \equiv \neg \forall_x \neg \mathbf{P}$ | $\mathbf{P} \vee \mathbf{Q} \equiv \neg (\neg \mathbf{P} \wedge \neg \mathbf{Q})$ |

❖ **Equality:-**

- First – order logic includes one more way of using equality symbol to make atomic sentences.
- Use of equality symbol

  - ✓ The equality symbol can be used to make statements to the effect that two terms refer to the same object.
  - ✓ For Example: - Father (John) = Henry says that the object referred to by Father (John) and the object referred to by Henry are the same.
  - ✓ Determining the truth of an equality sentence is simply a matter of seeing that the referents of the two terms are the same object.
  - ✓ The equality symbol can be used to state facts about a given function

- ✓ It can also be used with negation to insist that two terms are not the same object.
- ✓ For Example:- To say that Richard has at least two brothers, write as

$$\exists_{x,y} \text{Brother}(x, \text{Richard}) \wedge \text{Brother}(y,\text{Richard}) \wedge \neg (x = y)$$

### 2.5.4 Using First – Order Logic

The best way to learn about FOL is through examples. In Knowledge representation, a domain is just some part of the world about which some knowledge is to be expressed.

❖ **Assertions:-**

- Sentences that are added to knowledge base using **TELL**, exactly as in propositional logic are called assertion (Declaration/Statement).
- For Example:- It can be declared that " John is a King and that Kings are persons"

TELL (KB, King(John))
TELL (KB, $\forall_x$ King(x) $\Rightarrow$ Person(x))

❖ **Queries:-**

- Questions of the knowledge base can be asked using **ASK.**
- For Example:- **ASK(KB, King(John))** returns **true**.
- Questions asked using **ASK** are called queries or goals.
- Generally speaking, any query that is logically needed by the knowledge base should be answered positively.
- For Example:- Given the two assertions in the preceding line, the query

**ASK(KB, Person(John)** should also return **true**

❖ **Substitution/Binding List:-**

- Substitution or Binding list is a set of variable/term pairs.
- It is a standard form for an answer of a query with existential variables.
- For Example:- "Is there an x such that…" is solved by providing such an x.
- Given Just the two assertions, the answer would be {x/John}
- If there is more than one possible answer, a list of substitutions can be returned.

❖ **The Kinship Domain:-**

- Kinship domain is the domain of family relationships, or Kinship.
- This domain includes facts such as "Elizabeth is the mother of Charles" and "Charles is the father of William" and rules such as "One's grandmother is the mother of one's parent".
- The objects in this domain are people.
- There will be two unary predicates as **"Male"** and **"Female"**
- Kinship relations will be represented by binary predicates.

- For Example:- parenthood, brotherhood, marriage and so on are represented by Parent, sibling, Brother, Sister, Child, Daughter,Son,Spouse,Wife, Husband, Grandparents, Grandchild, Cousin, Aunt, and Uncle.
- For Example:-
- **One's mother is One's female parent:**

$$\forall m, c \; Mother(c) = m \Leftrightarrow Female(m) \wedge Parent(m,c)$$

❖ **Axioms:-**

- Axioms are commonly associated with purely mathematical domains.
- The axioms define, the Mother function and the Husband, Male, Parent and Sibling predicates in terms of other predicates.
- They provide the basic factual information from which useful conclusions can be derived.
- Kinship axioms are also definitions : they have the form $\forall_{x,y}$ **p(x,y)** $\Leftrightarrow$ ...

❖ **Theorems:-**

- Not all logical sentences about a domain are axioms.
- Some are Theorems-that is, they are caused by the axioms.
- For Example:-

$$\forall_{x,y} \; Sibling(x,y) \Leftrightarrow ... Sibling(y,x)$$

- The above declaration that siblinghood is symmetric
- It's a theorem that follows logically from the axiom that defines siblinghood.
- If ASK Questions the knowledge base this sentence, it should return true
- From logical point of view, a knowledge base need contain only axioms and no theorems
- From a practical point of view, theorems are essential to reduce the computational cost of deriving new sentences.

❖ **Numbers:-**

- Numbers are perhaps the most brilliant example of how a large theory can be built up from a tiny heart of axioms.
- Requirements
  - ✓ A predicate ***NatNum*** is needed that will be true of natural numbers
  - ✓ One constant symbol, 0
  - ✓ One function symbol, S (Successor)

- **Peano Axioms:-**
  - ✓ The peano axioms define natural numbers and addition.
  - ✓ Natural numbers are defined recursively:

NatNum (0)

$$\forall_n \text{ NatNum } (n) \Rightarrow \text{NatNum } (S(n))$$

- ✓ That is, 0 is a natural number, and for every object n, if n is a natural number then S(n) is a natural number,
- ✓ So the natural numbers are 0, S(0), S(S(0)), and so on..

❖ **Sets:-**

- The domain of sets is also fundamental to mathematics as well as to commonsense reasoning
- The empty set is a constant written as { }.
- There is one unary predicate, Set, which is true of sets.
- The binary predicates are x € s (**x** is a member of set **s**) and s1 ⊆ s2 (set s1 is a subset, not necessarily proper, of set s2)
- The binary functions are s1 ∩ s2 (the intersection of two sets), s1 ∪ s2 (the union of two sets), and {x/s} (the set resulting from adjoining element x to set **s**)
- One possible set of axioms is as follows,

  - ✓ The only sets are the empty set and those made by adjoining something to a set

    $$\forall_s \textbf{ Set(s)} \Leftrightarrow \textbf{(s = \{\}) V } (\exists_{x,} \textbf{ s2 Set(s2) } \Lambda \textbf{ s = \{x/s2\}}$$

  - ✓ There is no way to decompose EmptySet into a smaller set and an element:

    $$\neg \exists_{x,s} \textbf{ \{x/s\} = \{ \}}$$

  - ✓ Adjoining an element already in the set has no effect:

    $$\forall_{x,s} \quad x \in \text{(set membership) } s \Leftrightarrow s = \{x/s\}$$

  - ✓ The only members of a set are the elements that were connected into it. This can be expressed recursively, saying that **x** is a member of **s** if and only if **s** is equal to some set S2 connected with some element **y**, where either **y** is the same as **x** or **x** is a member of S2

    $$\forall_{x,s} \quad x \in s \Leftrightarrow [\exists_{y,} \, s2 \, (s=\{y/s2\} \Lambda (x=y \text{ V } x \in s2)) ]$$

  - ✓ A set is subset of another set if and only if all of the first sets members are members of the second set

    $$\forall_{s1,s2} \quad s1 \subseteq s2 \Leftrightarrow (\forall_x x \in s1 \Rightarrow x \in s2)$$

  - ✓ Two sets are equal if and only if each is a subset of the other

    $$\forall_{s1,s2} \quad (s1 = s2) \Leftrightarrow (s1 \subseteq s2 \Lambda s2 \subseteq s1)$$

- ✓ An object is in the Intersection of two sets if and only if it is a member of both sets

$$\forall_{x, s1, s2} \quad x \in (s1 \cap s2) \quad (x \in s1 \wedge x \in s2)$$

- ✓ An object is in the union of two sets if and only if it is a member of either set

$$\forall_{x, s1, s2} \quad x \in (s1 \cup s2) \quad (x \in s1 \vee x \in s2)$$

❖ **Lists:-**

- Lists are similar to sets.
- The differences are that lists are ordered and the same element can appear more than once in a list.
- **Nil** is the constant list with no elements
- **Cons, Append, First,** and **Rest** are functions.
- **Find** is the predicate that does for lists what ***Member*** does for sets.
- **List?** is a predicate that is true only of lists.
- The empty list is **f1.**
- The term **Cons (x, y),** where y is a nonempty list, is written **[x/y]**.
- The term **Cons (x, Nil)**, (i.e. The list containing the element **x**), is written as **x1.**
- A list of several elements, such as [A,B,C] corresponds to the nested term **Cons(A, Cons(B, Cons(C, Nil)))**

❖ **The Wumpus World:-**

- The first order axioms of wumpus world are more concise, capturing in a natural way what exactly we want to represent the concept.
- Here the more interesting question is **"how an agent should organize what it knows in order to take the right actions".**
- For this purpose we will consider three agent architectures:

  - ✓ Reflex agents - classify their percept and act accordingly
  - ✓ Model based agents - construct an internal representation of the World and use it to act
  - ✓ Goal based agents - form goals and try to achieve them

- The first step of wumpus world agent construction is to define the interface between the environment and the agent.
- The percept sentence must include both the percept and the time at which it occurred, to differentiate between the consequent percepts.
- For Example:-

**Percept ([Stench, Breeze, Glitter, None, None], 3)**

- In this sentence

  **Percept** - **predicate**

**Stench, Breeze, Glitter** - **Constants**
**3** - **Integer to represent to time.**

- The agents action are,
  **Turn (Right)**
  **Turn (Left)**
  **Forward**
  **Shoot**
  **Grab**
  **Release**
  **Climb**
- To determine which is best, the agent program constructs a query such as
  $\exists_a$ **BestActions(a,5)**
- **ASK** solves this query and returns a binding list such as {a/Grab}.
- The agent program then calls **TELL** to record the action which was taken to update the Knowledge base **KB.**

❖ **Types of Sentences:-**

- The percept sentences are classified in to two as,
  - ✓ **Synchronic (Same time)**
  - ✓ **Diachronic (across time)**
- **Synchronic: -** The sentences dealing with time is synchronic if they relate properties of a world state to other properties of the same world state.
- **Diachronic: -** The sentences describing the way in which the world changes (or does not change) are diachronic sentences

❖ **Kinds of Synchronic Rules:-**

- There are two kinds of synchronic rules that could allow to capture the necessary information for deductions are,
  - ✓ Diagnostic rules
  - ✓ Casual rules
  - o **Diagnostic Rules: -** Infer the presence of hidden properties directly from the percept – derived (observed) information.
  - o For Example: - For finding pits, if a square is breezy, some adjacent square must contain a pit.

$$\forall_s \textbf{Breezy(s)} \Rightarrow \exists_r \textbf{Adjacent (r,s)} \Lambda \textbf{Pit(r)}$$

  - o If a square is not breezy, no adjacent square contains a pit.

$$\forall_s \neg\textbf{Breezy(s)} \Rightarrow \neg\exists_r \textbf{Adjacent (r,s)} \Lambda \textbf{Pit(r)}$$

  - o Combining these two, the derived biconditional sentence is :

$$\forall_s \textbf{Breezy(s)} \Leftrightarrow \exists_r \textbf{Adjacent (r,s)} \Lambda \textbf{Pit(r)}$$

- **Casual Rules: -** Reflect the assumed direction of casuality in the world. Some hidden property of the world causes certain percepts to be generated.
- For Example:- A Pit causes all adjacent squares to be breezy.

$$\forall_r \textbf{Pit(r)} \Rightarrow [\forall_s \textbf{Adjacent (r, s)} \Rightarrow \textbf{Breezy(s)}]$$

- If all squares adjacent to a given square are pitless, the square will not be breezy

$$\forall_s [\forall_r \textbf{Adjacent (r,s)} \Rightarrow \neg \textbf{Pit(r)}] \Rightarrow \neg \textbf{Breezy(s)}$$

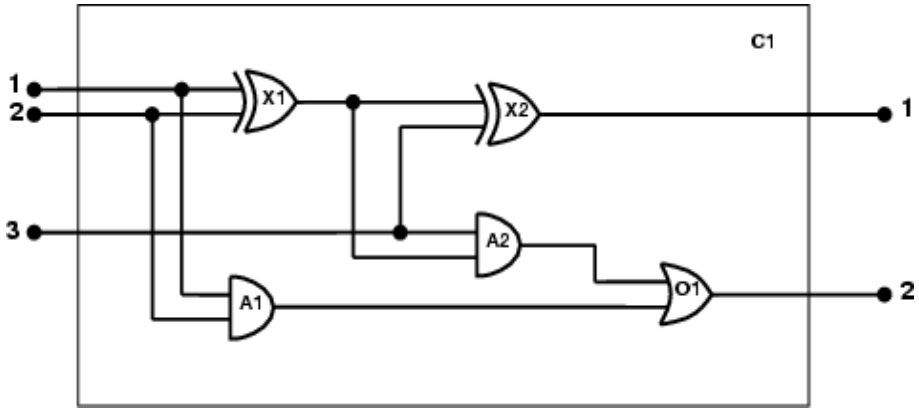
- System that reason with casual rules are called model-based reasoning systems, because the casual rules from a model of how the environment operates.

### 2.5.5 KNOWLEDGE ENGINEERING IN FIRST ORDER LOGIC:-

- **Knowledge Engineering: -** The general process of Knowledge Base **KB** Construction.
- **Knowledge Engineer: -** Who investigates a particular domain, learns what concepts are important in that domain and creates a formal representation of the objects and relations in the domain.
- The knowledge engineering projects vary widely in content, scope and difficulty, but all projects include the following steps,

  - ✓ **Identify the Task: -** The Knowledge engineer should identify the **PEAS** description of the domain.
  - ✓ **Assemble the relevant knowledge: -** The idea of combining expert's knowledge of that domain (i.e.) a process called **knowledge acquisition.**
  - ✓ **Decide on a vocabulary of predicates, functions and constants: -** Translate the important domain-level concepts into logical level name. The resulting vocabulary is known as **ontology** of the domain, which determines what kinds of things exist, but does not determine their specific properties and interrelationships.
  - ✓ **Encode general knowledge about the domain: -** The knowledge engineer writes the axioms (rules) for all the vocabulary terms. The misconceptions are clarified from step 3 and the process is iterated.
  - ✓ **Encode a description of the specific problem instance: -** To write simple atomic sentences about instances of concepts that are already part of the ontology.
  - ✓ **Pose queries to the inference procedure and get answers: -** For the given query the inference procedure operate on the axioms and problem specific facts to derive the answers.
  - ✓ **Debug the knowledge base: -** For the given query , if the result is not a user expected on then **KB is** updated with relevant or missing axioms.

- The seven step process is explained with the domain of **ELECTRONIC CIRCUITS DOMAIN.**




- ✓ **Identify the task: -**
  - Analyse the circuit functionality, does the circuit actually add properly? (Circuit Verification).
- ✓ **Assemble the relevant knowledge: -**
  - The circuit is composed of wire and gates.
  - The four types of gates (AND, OR, NOT, XOR) with two input terminals and one output terminal knowledge is collected.
- ✓ **Decide on a vocabulary: -**
  - The functions, predicates and constants of the domain are identified.
  - Functions are used to refer the type of gate.
    Type (x1) = XOR,
    where x1 ---- Name of the gate and Type ----- function
  - The same can be represented by either binary predicate (or) individual type predicate.
    Type (x1, XOR) – binary predicate
    XOR (x1) – Individual type
  - A gate or circuit can have one or more terminals. For x1, the terminals are x1In1, x1In2 and x1 out1
    Where x1 In1 ------ 1st input of gate x1
    x1 In2------ 2nd input of gate x1
    x1 out1 ----- output of gate x1
  - Then the connectivity between the gates represented by the predicate connected. (i.e.) connected (out(1, x1), In(1,x2)).
  - Finally the possible values of the output terminal C1, as true or false, represented as a signal with 1 or 0.
- ✓ **Encode general knowledge of the domain:-**
  - This example needs only seven simple rules to describe everything need to know about circuits
  - If two terminals are connected, then they have the same signal:
    - $\forall t_1,t_2$ Connected$(t_1, t_2) \Rightarrow$ Signal$(t_1)$ = Signal$(t_2)$
  - The signal at every terminal is either 1 or 0 (but not both):
    - $\forall t$ Signal$(t) = 1 \vee$ Signal$(t) = 0$
    - $1 \neq 0$

- Connected is a commutative predicate
  - $\forall t_1,t_2$ Connected($t_1$, $t_2$) $\Rightarrow$ Connected($t_2$, $t_1$)
- An OR gate's output is 1 if and only if any of its input is 1:
  - $\forall g$ Type(g) = OR $\Rightarrow$ Signal(Out(1,g)) = 1 $\Leftrightarrow \exists n$ Signal(In(n,g)) = 1
- An AND gate's output is 0 if and only if any of its inputs is 0
  - $\forall g$ Type(g) = AND $\Rightarrow$ Signal(Out(1,g)) = 0 $\Leftrightarrow \exists n$ Signal(In(n,g)) = 0
- An XOR gate's output is 1 if and only if inputs are different:
  - $\forall g$ Type(g) = XOR $\Rightarrow$ Signal(Out(1,g)) = 1 $\Leftrightarrow$ Signal(In(1,g)) $\neq$ Signal(In(2,g))
- A NOT gate's output is different from its input:
  - $\forall g$ Type(g) = NOT $\Rightarrow$ Signal(Out(1,g)) $\neq$ Signal(In(1,g))

✓ **Encode the specific problem instance:**

- First, we categorize the gates:

  Type($X_1$) = XOR  Type($X_2$) = XOR

  Type($A_1$) = AND  Type($A_2$) = AND

  Type($O_1$) = OR

- Then, we show the connections between them

Connected(Out(1,$X_1$),In(1,$X_2$))  Connected(In(1,$C_1$),In(1,$X_1$))

Connected(Out(1,$X_1$),In(2,$A_2$))  Connected(In(1,$C_1$),In(1,$A_1$))

Connected(Out(1,$A_2$),In(1,$O_1$))  Connected(In(2,$C_1$),In(2,$X_1$))

Connected(Out(1,$A_1$),In(2,$O_1$))  Connected(In(2,$C_1$),In(2,$A_1$))

Connected(Out(1,$X_2$),Out(1,$C_1$))  Connected(In(3,$C_1$),In(2,$X_2$))

Connected(Out(1,$O_1$),Out(2,$C_1$))  Connected(In(3,$C_1$),In(1,$A_2$))

✓ **Pose Queries to the inference procedure:**

- What combinations of inputs would cause the first output of C1(the sum bit) to be 0 and the second output of C1 (the carr y bit) to be 1?

  $\exists i_1,i_2,i_3$ Signal(In(1,C1)) = $i_1 \wedge$ Signal(In(2,$C_1$)) = $i_2 \wedge$ Signal(In(3,$C_1$)) = $i_3 \wedge$ Signal(Out(1,$C_1$)) = o $\wedge$ Signal(Out(2,$C_1$)) = 1

- The answers are substitutions for the variables $i_1,i_2$ and $i_3$ Such that the resulting sentence is entailed by the knowledge base.

- There are three such substitutions as:

  { $i_1$/1,$i_2$/1 , $i_3$/0} { $i_1$/1,$i_2$/0 , $i_3$/1} { $i_1$/0,$i_2$/1 , $i_3$/1}

- What are the possible sets of values of all the terminals for the adder circuit?

  $\exists i_1,i_2,i_3,o_1,o_2$ Signal(In(1,C1)) = $i_1 \wedge$ Signal(In(2,$C_1$)) = $i_2 \wedge$ Signal(In(3,$C_1$)) = $i_3 \wedge$ Signal(Out(1,$C_1$)) = $o_1 \wedge$ Signal(Out(2,$C_1$)) = $o_2$

✓ **Debug the knowledge base:**

- The knowledge base is checked with different constraints.
- For Example:- if the assertion $1 \neq 0$ is not included in the knowledge base then it is variable to prove any output for the circuit, except for the input cases 000 and 110.

## 2.6 Inference in First-order Logic:-

- We have learned seven inference rules of propositional logic.
- These rules are applicable for First-order logic also
- With those rules First-order logic holds some additional rules "with quantifiers" in which substituting particular individual for the variable is done (i.e.) SUBST(Θ, α) to denote the result of applying the substitution (or) binding list Θ to the sentence α.
- For Example:-
  SUBST ( {x/Ram, y/John} Likes(x, y)) = Likes(Ram, John)
- The following are the new three inference rules for First-order Logic.
  - ✓ Universal Elimination
  - ✓ Existential Elimination
  - ✓ Existential Introduction
- **Universal Elimination:-** For any sentence α, variable v and ground term g;

  **$\forall$v α / SUBST( {v/g}, α)**
  Example:-
  $\forall$x likes (x, Icecream) is a sentence **α**.

SUBST ({x/John}, α) is a substitution Θ = John
Likes (John, Icecream) – Inferred sentence

- **Existential Elimination :-** For any sentence α variable v, and constant symbol k that does not appear elsewhere in the Knowledge base:

**∃v α / SUBST( {v/k}, α)**
Example:-
∃x Kill (x, Victim) – **α**
SUBST({x/Murderer, Victim}, **α**) where **Θ** = Murderer
Kill (Murderer, Victim) – Inferred Sentence

- **Existential Introduction :-** For any sentence α, variable v that does not occur in **α**, and ground term **g** that does occur in **α**

**α / ∃v SUBST( {g/v}, α)**
Example:-
Likes (John, Icecream) – **α**
∃x Likes (x, Icecream) – Inferred Sentence

### 2.6.1 AN EXAMPLE PROOF USING FIRST- ORDER LOGIC:-

- An application of inference rule is matching their premise patterns to the sentences in the KB and then adding their conclusion patterns to the KB.
- **Task: -** For the given situation described in English, Convert it into its equivalent FOL representation and prove that "West is a Criminal".
- **Situation: -** The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American.
- **Solution: -** The given description is splitted into sentences and is converted into its corresponding FOL representation.

  - ✓ It is a crime for an American to sell weapons to hostile nations:
    **∀x y z American(x) ∧ Weapon(y) ∧ Nation(z) ∧ Hostile(z) ∧ Sells(x, y, z ⇒ Criminal(x)**
  - ✓ Nono … has some missiles,
    **∃x Owns(Nono,x) ∧ Missile(x)**
  - ✓ all of its missiles were sold to it by Colonel West
    **∀x Missiles(x) ∧ Owns(Nono,x) ⇒ Sells(West,x,Nono)**
  - ✓ We will also need to know that missiles are weapons
    **∀x Missile(x) ⇒ Weapon(x)**
  - ✓ An enemy of America counts as "hostile"
    **∀x Enemy(x,America) ⇒ Hostile(x)**
  - ✓ West, who is American …
    **American(West)**

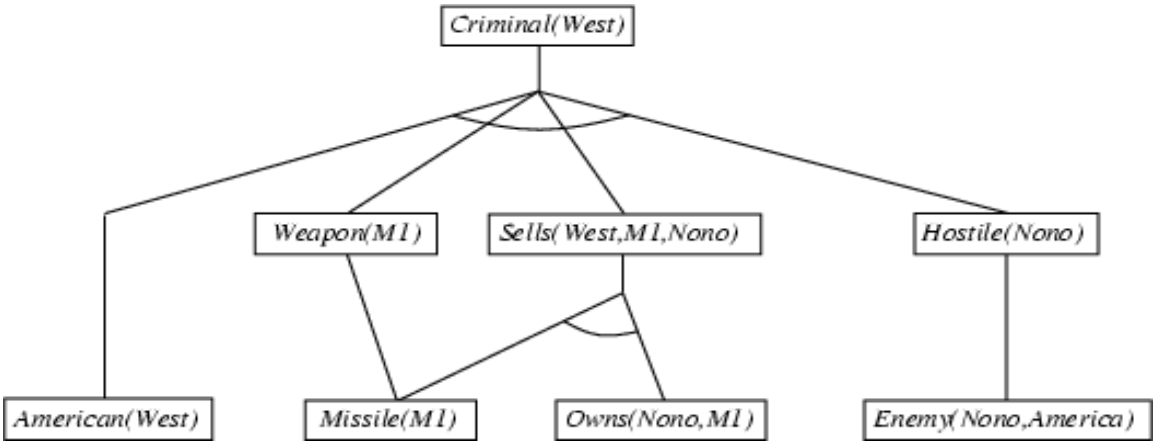
- ✓ Nono, is a nation
  **Nation (Nono)**
- ✓ Nono, an enemy of America
  **Enemy (Nono, America)**
- ✓ America is nation
  **Nation (America)**

## 2.7 Forward Chaining:-

- The Generalized Modus Ponens rule can be used by Forward Chaining.
- From the sentences in the **KB** which in turn derive new conclusions.
- Forward chaining is preferred when new fact is added to the database and we want to generate its consequences.
- **Forward Chaining Algorithm:-**
  - ✓ Forward chaining is triggered by the addition of new fact "p" into the knowledge base (i.e.) the action **TELL** is performed.
  - ✓ If the new fact is a rename of any other existing sentence in the **KB** then it is not included in **KB.**
  - ✓ With the new fact "p" find all premises that had "p" as premise and if any other premise is already known to hold then its consequence is included in **KB.**
  - ✓ The important operation of forward chaining is renaming : One sentence is a renaming of another if, they are identical except for the names of the variables.
  - ✓ **For Examples:-**
    - Likes(x, Icecream) and Likes(y, Icecream) are renaming of each other.
    - Likes(x,x) and Likes(x,y) are not renaming of each other (i.e.) its variable differs, the meaning is logically different.
  - ✓ Consider the **KB** of crime problem represented in Horn form to explain the concept of forward chaining.
  - ✓ The implication sentences are (i), (iv), (v), (vi)
  - ✓ Two iterations are required:
  - ✓ On the first iteration,
    - Step (i) has unsatisfied premises
    - Step (iv) is satisfied with {x/M1} and sells (west, M1,Nono) is added
    - Step (v) is satisfied with {x/m1} and weapon (M1) is added
    - Step (vi) is satisfied with {x/Nono}, and Hostile (Nono) is added
  - ✓ On the second iteration,
    - Step (i) is satisfied with {x/West, y/M1, z/Nono} and Criminal(west) is added.
  - ✓ The following table shows the forward chaining algorithm,
  - ✓ **Inputs:- KB**, the Knowledge base, a set of first-order definite clauses **α,** the query, an atomic sentence
  - ✓ **Local variable:- new,** the new sentences inferred on each iteration

```
function FOL-FC-ASK(KB, α) returns a substitution or false
  repeat until new is empty
    new ← { }
    for each sentence r in KB do
      (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-APART(r)
      for each θ such that (p1 ∧ ... ∧ pn)θ = (p'1 ∧ ... ∧ p'n)θ
                  for some p'1, ..., p'n in KB
        q' ← SUBST(θ, q)
        if q' is not a renaming of a sentence already in KB or new then do
          add q' to new
          φ ← UNIFY(q', α)
          if φ is not fail then return φ
    add new to KB
  return false
```

- ✓ The following figure shows the proof tree generated by forward chaining algorithm on the Crime Example,




- ✓ The above discussed inference processes are not directed towards solving any particular problem: for this reason it is called a **data-driven or data-directed procedure.**
- ✓ In this problem, no new inferences are possible because every sentence concluded by forward chaining is already exist in the **KB.** Such a **KB** is called a **fixed point** of the inference process.
- ✓ FOL-FC-ASK function is sound and complete.
- ✓ Every inference is an application of generalized modus ponens, which is sound.
- ✓ Then it is complete for definite clauses **KB** (i.e.) it answers every query whose answers are entailed by any **KB** of definite clauses.

### 2.7.1 Efficient Forward chaining:-

- The above discussed FC Algorithm has three possible types of complexity.

- ✓ **Pattern Matching: -** "inner loop" of the algorithm involves finding all possible unifiers such that the premise of a rule unifies with a suitable set of facts in the **KB.**
- ✓ **Matching rules against known facts:-** The algorithm re-checks every rule on every iteration to see whether its premises are satisfied, even if very few additions are made to the **KB** on each iteration
- ✓ Irrelevant facts to the goal are generated

- In forward chaining approach, inference rules are applied to the knowledge base, yielding new assertions.
- This process repeats forever or until some stopping criterian is met.
- This method is appropriate for the design of an agent, that is on each cycle, we add the percepts to the knowledge base and run the forward chainer, which chooses an action to perform according to a set of condition action rules.
- Theoretically a **production system** can be implemented with a theorem prover, using resolution to do forward chaining over a full first order knowledge base.
- An efficient language can be used to perform this task, because it reduces the branching factor.
- The typical production system has three features:
  - ✓ The system maintained a **KB** called the working memory which has a set of positive literals with no variable
  - ✓ The system maintains a **rule memory**. This contains a set of inference rules $P_1 \wedge P_2 \Rightarrow act_1 \wedge act_2 \ldots$. That $act_i$ is executed when $p_i$ is satisfied, which performs adding or deleting an element from the **working memory – match phase.**
  - ✓ In each cycle, the system computes the subset of rules whose left-hand side is satisfied by the current contents of the **working memory - match phase.**

## 2.8 Backward Chaining:-

- Backward chaining is designed to find all answers to a question asked to the knowledge base. Therefore it requires a **ASK** procedure to derive the answer.
- The procedure **BACK WARD-CHAIN** will check two constraints.
  - ✓ If the given question can derive a answer directly from the sentences of the knowledge base then it returns with answers.
  - ✓ If the first constraint is not satisfied then it finds all implications whose conclusion unifies with the query and tries to establish the premises of those implications. If the premise is a conjunction then BACK-CHAIN processes the conjunction conjunct by conjunct, building up the unifiers for the whole premises as it goes.
- **Composition of Substitutions:-** COMPOSE($\Theta_1$, $\Theta_2$) is the substitution whose effect is identical to the effect of applying each substitution in turn (i.e.),

  **SUBST (COMPOSE ($\Theta_1$, $\Theta_2$), p) = SUBST ($\Theta_2$, SUBST ($\Theta_1$, p))**
- For Example:-

  **P – Sells (x,M1, y)**

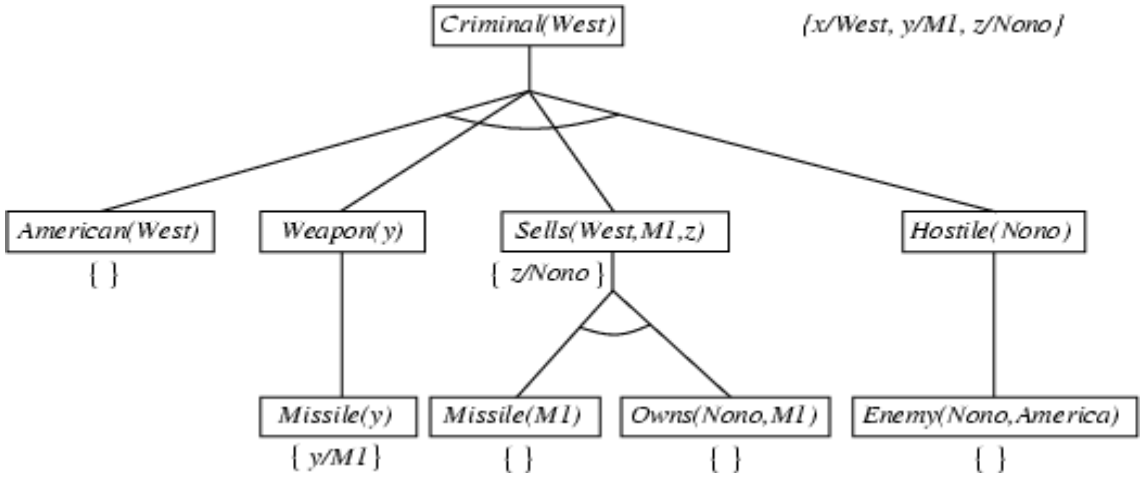
**SUBST ($\Theta_2$, SUBST ($\Theta_1$, p))**
**SUBST ($\Theta_2$, SUBST (x/West, p)) i.e. ($\Theta_1$ = x/west)**
**SUBST ((y/Nono), (x/West, p)) i.e. ($\Theta_2$ = y/Nono)**
**Therefore p – Sells (West, M1, Nono)**

- The following table shows the backward chaining algorithm,

**function** FOL-BC-ASK($KB$, goals, $\theta$) **returns** a set of substitutions
  **inputs**: $KB$, a knowledge base
    *goals*, a list of conjuncts forming a query
    $\theta$, the current substitution, initially the empty substitution { }
  **local variables**: *ans*, a set of substitutions, initially empty

  **if** *goals* is empty **then return** $\{\theta\}$
  $q' \leftarrow$ SUBST($\theta$, FIRST(*goals*))
  **for each** $r$ **in** $KB$ where STANDARDIZE-APART($r$) = $(p_1 \wedge \ldots \wedge p_n \Rightarrow q)$
    and $\theta' \leftarrow$ UNIFY($q, q'$) succeeds
    *ans* $\leftarrow$ FOL-BC-ASK($KB$, $[p_1, \ldots, p_n | \text{REST}(goals)]$, COMPOSE($\theta, \theta'$)) $\cup$ *ans*
  **return** *ans*

- The following graph shows the proof tree to infer that west is a criminal,




- To prove Criminal(x), we have to prove the five conjuncts below it
- Some of which are directly exist in the knowledge base, others require one more iteration of backward chaining.
- In the search process the substitution of values for the variables has to be done in a correct way, otherwise it may lead to failure solution.
- The following are the some properties of Backward Chaining,
  - Depth-first recursive proof search: space is linear in size of proof
  - Incomplete due to infinite loops
    - $\Rightarrow$ fix by checking current goal against every goal on stack

- Inefficient due to repeated subgoals (both success and failure)
  - $\Rightarrow$ fix using caching of previous results (extra space)
- Widely used for logic programming

### 2.8.1 Logic Programming:-

- A system in which KB can be constructed and used.
- A relation between logic and algorithm is summed up in Robert Kowalsh equation

  **Algorithm = Logic + Control**
- Logic programming languages, usually use backward chaining and input/output of programming languages.
- A logic programming language makes it possible to write algorithms by augmenting logic sentences with information to control the inference process.
- For Example:- PROLOG
  - ✓ A prolog program is described as a series of logical assertions each of which is a Horn Clause.
  - ✓ A Horn Clause is a Clause that has atmost one positive literal,

    Example: - P, $\neg P \wedge Q$
  - ✓ Implementation: - All inferences are done by backward chaining, with depth first search. The order of search through the conjuncts of an antecedent is left to right and the clauses in the KB are applied first-to- last order.
- **Example for FOL to PROLOG conversion:-**
  - **FOL**
    - ✓ $\forall$**x Pet(x)** $\wedge$ **Small(x)** $\Rightarrow$ **Apartment(x)**
    - ✓ $\forall$**x Cat(x) v Dog(x)** $\Rightarrow$ **Pet(x)**
    - ✓ $\forall$**x Product(x)** $\Rightarrow$ **Dog(x)** $\wedge$ **Small(x)**
    - ✓ **Poodle(fluffy)**
  - **Equivalent PROLOG representation**
    - ✓ **Apartment(x) :- Pet(x), Small(x)**
    - ✓ **Pet(x) :- Cat(x)**
      **Pet(x) :- Dog(x)**
    - ✓ **Dog(x) :- Poodle(x)**
      **Small(x) :- Poodle(x)**
    - ✓ **Poodle(fluffy)**
  - In the PROLOG representation the consequent or the left hand side is called as head and the antecedent or the right hand side is called as **body.**
- **Execution of a PROLOG program:-**
  - The execution of a prolog program can happen in two modes,
    1. Interpreters
    2. Compilers
  - **Interpretation:**
    - ✓ A method which uses **BACK-CHAIN** algorithm with the program as the **KB.**

- ✓ To maximize the speed of execution, we will consider two different types of constraints executed in sequence, They are
    1. **Choice Point: -** Generating sub goals one by one to perform interpretation.
    2. **Trail: -** Keeping track of all the variables that have been bound in a stack is called as trail.

- **Compilation:-**
    - ✓ Procedure implementation is done to run the program (i.e.) calling the inference rules whenever it is required for execution.

**2.9 Unification:-**

- The process of finding all legal substitutions that make logical expressions look identical and
- Unification is a "pattern matching" procedure that takes two atomic sentences, called literals, as input, and returns "failure" if they do not match and a substitution list, Theta, if they do match.
- Theta is called the most general unifier (mgu)
- All variables in the given two literals are implicitly universally quantified
- To make literals match, replace (universally-quantified) variables by terms
- The unification routine, UNIFY is to take two atomic sentences p and q and returns α substitution that would make p and q look the same

  UNIFY (p, q) = θ where SUBST (θ, p) = SUBST (θ, q)

  θ = Unifier of two sentences
- For example:-

  p – S1(x, x) q

  – S1(y, z)

  Assume θ = y

  p – S1(y, y) – x/y (Substituting y for x)

  q – S1(y, y) – z/y (Substituting y for z)
- In the above two sentences (p, q), the unifier of two sentences (i.e.) θ = y is substituted in both the sentences, which derives a same predicate name, same number of arguments and same argument names.
- Therefore the given two sentences are unified sentences.
- The function UNIFY will return its result as fail, for two different types of criteria's as follows,
    - ✓ If the given two atomic sentences (p, q) are differs in its predicate name then the UNIFY will return failure as a result

      For Example: - p – hate (M, C), q – try (M, C)
    - ✓ If the given two atomic sentences (p, q) are differs in its number of arguments then the UNIFY will return failure as a result

      For Example: - p – try (M, C), q – try (M, C, R)
- For Example: - The Query is **Knows (John, x) whom does John Know?**

- Some answers to the above query can be found by finding all sentences in the KB that unify with knows (John, x)
- Assume the KB has as follows,
  Knows (John, John)
  Knows (y, Leo)
  Knows (y, Mother(y))
  Knows (x, Elizabeth)
- The results of unification are as follows,
  UNIFY (knows (john, x), knows (John, Jane)) = {x/Jane}
  UNIFY (knows (john, x), knows (y, Leo)) = {x/Leo, y/John}
  UNIFY (knows (john, x), knows (y, Mother(y)) = {y/John, x/Mother (John)}
  UNIFY (knows (john, x), knows (x, Elizabeth)) = fail
- x cannot take both the values John and Elizabeth at the same time.
- Knows (x, Elizabeth) means "Everyone knows Elizabeth" from this we able to infer that John knows Elizabeth.
- This can be avoided by using standardizing apart one of the two sentences being unified (i.e.) renaming is done to avoid name clashes.
- For Example:-
  UNIFY (Knows (john, x), knows ($x_1$, Elizabeth)) = {x/Elizabeth, $x_1$/John}

### 2.9.1 Most general Unifier (MGU):-

- UNIFY should return a substitution that makes the two arguments look the same, but there may be a chance of more than one unifier.
- For Example:-
  UNIFY (knows (john, x), knows (y, z)) = {y/John, x/z} or {y/John, x/John, z/John}
- The result of applying 1st unifier is knows (John, z) and the 2nd unifier is knows (John, John).
- Here the first unifier result is more general than the second one, because it places less restriction on the values of the variables.
- This indicates that every unifiable pair of expressions, a single MGU is exist until renaming of variables.
- The following table shows the unification algorithm,
- The following are the steps to be done for unification of two sentences p and q is given below,
  - ✓ Recursively explore the two expressions simultaneously along with unifier returns failure if two corresponding points in the structure do not match. Therefore the time complexity is $O(n^2)$ in the size of the expressions being unified.
  - ✓ When the variable is matched against a complex term, one must check, whether the variable itself occurs, if it is true then returns failure (consistent unifier is not allowed) – occur check.

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound
          y, a variable, constant, list, or compound
          θ, the substitution built up so far

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(ARGS[x], ARGS[y], UNIFY(OP[x], OP[y], θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(REST[x], REST[y], UNIFY(FIRST[x], FIRST[y], θ))
  else return failure
```

```
function UNIFY-VAR(var, x, θ) returns a substitution
  inputs: var, a variable
          x, any expression
          θ, the substitution built up so far

  if {var/val} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

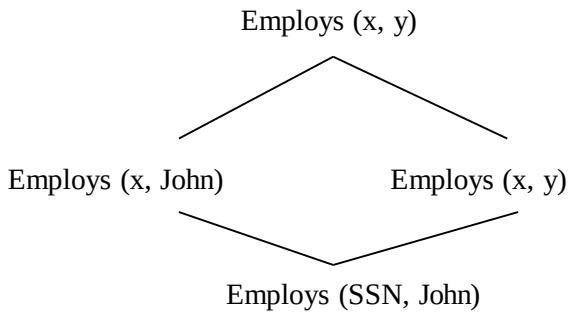
- The above table shows the unification algorithm

### 2.9.2 Storage and retrieval:-

- Once the data type for sentences and terms are defined, we need to maintain asset of sentences in a KB (i.e.) to store and fetch a sentence or term,
  - ✓ Store (s) – stores a sentence s.
  - ✓ Fetch (q) – returns all unifiers
- Such that the query q unifies with some sentences in the KB.
- For Example: - The unification for Knows (John, x) is an Instance of fetching.
- The simplest way to store and fetch is maintain a long list in sequential order.
- For a Query q, call UNIFY (q, s) for every sentences s in the list, requires O(n) time on an n-element KB.
- To make the fetch more efficient, indexing the facts in KB is done.
- The different types of indexing are as follows,
  - ✓ Table based Indexing
  - ✓ Tree based Indexing

- ✓ Predicate based Indexing

- A simple form of indexing is predicate indexing puts all the knows facts in one bucket and all the brother facts in another.
- The buckets are stored in hash table for efficient access,
- Where hash table means "a data structure for storing and retrieving information indexed by fixed keys"
- Given sentence to be stored, it is possible to construct indices for all possible queries that unify with it,
- For Example:- For the fact Employs (SSN, John) the queries are as follows,
  - ✓ Employs (SSN, John) Does SSN employ John?
  - ✓ Employs (x, John) who employs John?
  - ✓ Employs (SSN, y) whom does SSN employ?
  - ✓ Employs (x, y) who employs whom?
- These queries form a substitution lattice (i.e.) Properties of Lattice:- child of any node in the lattice is obtained from its parent by a single substitution and the highest common descendent of any two nodes is the result of applying their most general unifier.
- For a predicate with n arguments, the lattice contains $O(2^n)$ nodes
- The following diagram shows the subsumption lattice,




### 2.9.3 Advantages and Disadvantages:-

**Advantages:-**

- The scheme works very well whenever the lattice contains a small number of nodes.
- For a predicate with n arguments, the lattice contains O(2n) nodes.

**Disadvantages:-**

- If function symbols are allowed, the number of nodes is also exponential in the size of the terms in the sentence to be stored. This can lead to a huge number of indices.
- At some point, the benefits of indexing are outweighed by the costs of storing and maintaining all the indices.

## 2.10 Resolution:-

- Resolution is a complete inference procedure for first order logic
- Any sentence a entailed by KB can be derived with resolution
- Catch: proof procedure can run for an unspecified amount of time

- At any given moment, if proof is not done, don't know if infinitely looping or about to give an answer
- Cannot always prove that a sentence a is *not* entailed by KB
- First-order logic is semidecidable

- Rules used in the resolution procedure are :
  - Simple resolution inference rule – premises have exactly two disjuncts

$$\frac{\alpha \vee \beta, \neg\beta \vee \gamma}{\alpha \vee \gamma} \quad \text{or equivalently} \quad \frac{\neg\alpha \Rightarrow \beta, \beta \Rightarrow \gamma}{\neg\alpha \Rightarrow \gamma}$$

  - Resolution inference rule – two disjunctions of any length, if one of disjunct in the class ($p_j$) unifies with the negation of the disjunct in the other class, then infer the disjunction of all the disjuncts except for those two,
  - **Using disjunctions:-** For literals $p_i$ and $q_i$, where UNIFY ($p_j$, $\neg q_k$) = $\theta$

$$\frac{p_1 \vee \ldots\ldots\ldots p_j \ldots\ldots\ldots \vee p_m, q_1 \vee \ldots\ldots\ldots q_k \ldots\ldots\ldots\ldots \vee q_n}{SUBST(\theta, (p_1 \vee \ldots\ldots\ldots p_{j-1} \vee p_{j+1} \ldots\ldots\ldots p_m \vee q_1 \ldots\ldots\ldots q_{k-1} \vee q_{k+1} \ldots\ldots\ldots\ldots \vee q_n}$$

  - **Using implications:-** For atoms $p_i$, $q_i$, $r_i$, $s_i$ where UNIFY ($p_j$, $q_k$) = $\theta$

$$p_1 \wedge \ldots\ldots\ldots p_j \wedge \ldots\ldots\ldots p_{n1} \Rightarrow r_1 \vee \ldots\ldots r_{n2}$$

$$\frac{s_1 \wedge \ldots\ldots\ldots s_{n3} \Rightarrow, q_1 \vee \ldots\ldots\ldots q_k \ldots\ldots\ldots\ldots q_{n4}}{SUBST(\theta, (p_1 \wedge \ldots p_{j-1} \wedge p_{j+1} \ldots\ldots p_{n1} \wedge s_1 \wedge \ldots\ldots s_{n3} \Rightarrow r_1 \vee \ldots\ldots\ldots r_{n2} \vee q_1 \vee \ldots\ldots\ldots\ldots q_{k-1} \vee q_{k+1} \ldots\ldots\ldots\ldots \vee q_{n4}}$$

### 2.10.1 Canonical Form (or) Normal form for Resolution:-

- The canonical form representation of sentences for resolution procedure (to derive pa proof) is done in two ways, they are as follows,
  - **Conjunctive Normal form (CNF):-** All the disjunctions are joined aqs one big sentences.
  - **Implicative Normal Form (INF):-** All the conjunctions of atoms on the left and a disjunction of atoms on the right.
- The following table shows the Knowledge base for CNF and INF,

| Conjuctive Normal Form | Implicative Normal Form |
|---|---|
| $\neg$ P(w) $\vee$ Q(w) | P(w) $\Rightarrow$ Q(w) |
| P(x) $\vee$ R(x) | True $\Rightarrow$ P(x) $\vee$ R(x) |
| $\neg$ Q(y) $\vee$ S(y) | Q(y) $\Rightarrow$ S(y) |
| $\neg$ R(z) $\vee$ S(z) | R(z) $\Rightarrow$ S(z) |

- Resolution is a generalization of Modus Ponen.
- The following is the representation of Modus Ponen rule in resolution as a special case (i.e.),

$$\frac{\alpha, \alpha \Rightarrow \beta}{\beta} \quad \text{is equivalent to} \quad \frac{True \Rightarrow \alpha, \alpha \Rightarrow \beta}{True \Rightarrow \beta}$$

### 2.10.2 Methods used for resolution technique

- **Resolution Proof:-** A set of inference rules and resolution rules can be used to derive a conclusion from the KB.
- **Resolution with refutation (or) proof by contradiction (or) reduction and absurdum:-** One complete inference procedure using resolution is refutation. The idea is to prove P, we assume P is false (i.e. add $\neg P$ to KB) and prove a contradiction, that is the KB implies P.

$$\mathbf{(KB \wedge \neg P \Rightarrow False) \Leftrightarrow (KB \Rightarrow P)}$$

- **Example:**

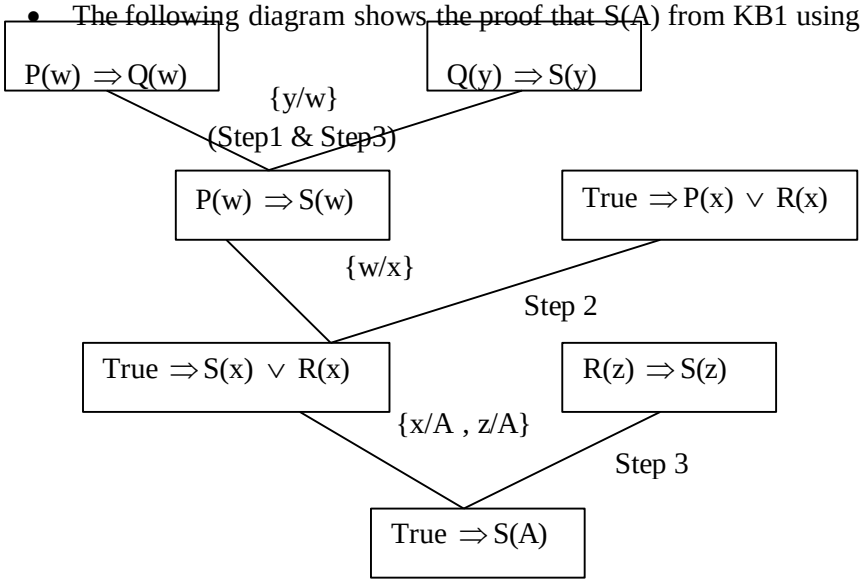
1. Prove that S(A) ts true from KB1 of CNF and INF representation using
   - ✓ Resolution Proof
   - ✓ Resolution with refutation

   **(a) Resolution Using INF representation:-**

   **Given (KB1):-**

   1. $P(w) \Rightarrow Q(w)$
   2. $True \Rightarrow P(x) \vee R(x)$
   3. $Q(y) \Rightarrow S(y)$
   4. $R(z) \Rightarrow S(z)$

- The following diagram shows the proof that S(A) from KB1 using resolution




- Resolution rule:- In the first step transitive implication rule is used
- Substitution of one predicate in the other. (i.e.) $P(x) \Rightarrow S(x)$ is substituted in $True \Rightarrow P(x) \vee S(x)$ that is instead of P(x), S(x) is substituted.

- Substitution of one predicate in the other. (i.e.) R(A) $\Rightarrow$ S(A) is substituted in True $\Rightarrow$ S(A) $\vee$ R(A) that is instead of R(A), S(A) is substituted, which derives True $\Rightarrow$ S(A) $\vee$ S(A) is equivalent to True $\Rightarrow$ S(A)
- Therefore S(A) is true is proved using resolution technique for INF representation.
- Where each "Vee" Shape in the proof tree represents a resolution step,
- The two sentences at the top are the premises from the KB, and the one at the bottom is the conclusion (or) resolvent.

**b. Resolution Using CNF representation:-**

**Given (KB1):-**

1. ~P(w) $\vee$ Q(w)
2. P(x) $\vee$ R(x)
3. ~Q(y) $\vee$ S(y)
4. ~R(z) $\vee$ S(z)

- **Resolution rule:-**

$$\frac{\alpha \vee \beta, \beta \vee \gamma}{\alpha \vee \gamma} \quad \text{(i.e.)} \quad \frac{\sim P(w) \vee Q(w), \sim Q(w) \vee S(w)}{\sim P(w) \vee S(w)}$$

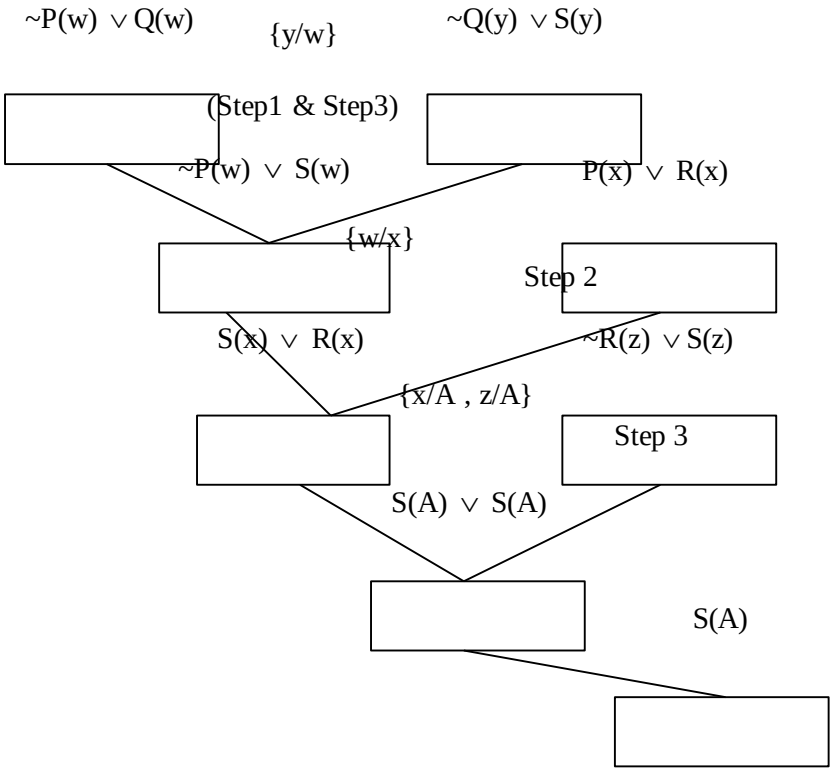
- **Resolution rule:-**

$$\frac{\alpha \vee \beta, \beta \vee \gamma}{\alpha \vee \gamma} \quad \text{(i.e.)} \quad \frac{R(x) \vee P(x), \sim P(x) \vee S(x)}{R(x) \vee S(x)}$$

- **Resolution rule:-**

$$\frac{\alpha \vee \beta, \beta \vee \gamma}{\alpha \vee \gamma} \quad \text{(i.e.)} \quad \frac{S(A) \vee R(A), \sim R(A) \vee S(A)}{S(A) \vee S(A)}$$

- Therefore S(A) is true is proved using resolution technique for CNF representation.
- The following diagram shows the proof that S(A) from KB1 using resolution.



**C. Resolution with refutation Proof using INF representation:-**

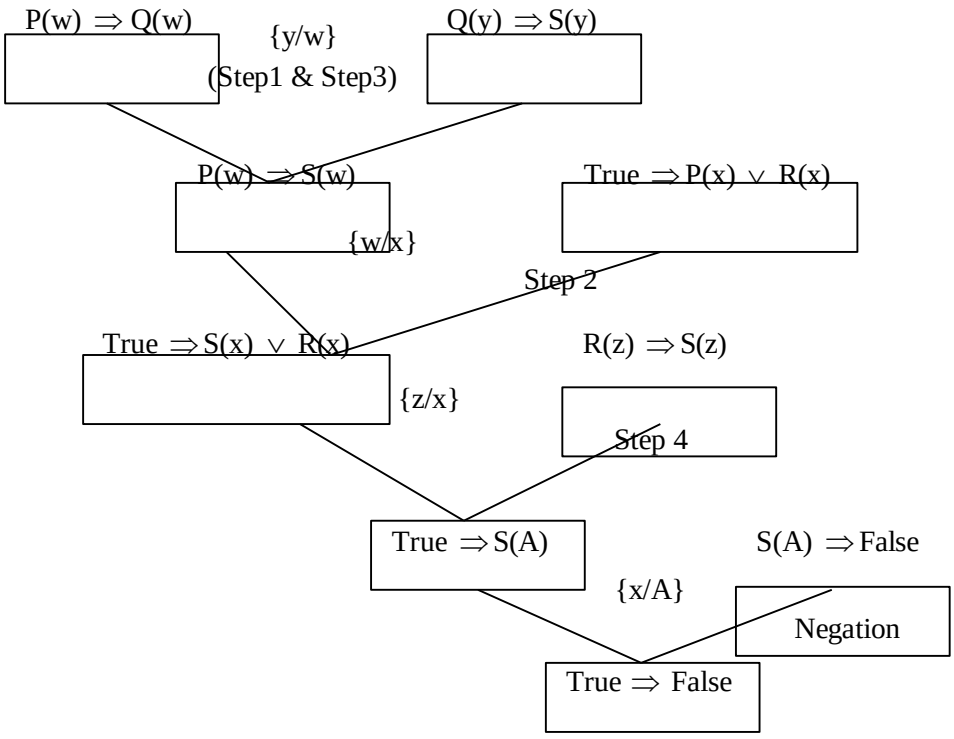
**Given (KB1):-**

1. $P(w) \Rightarrow Q(w)$
2. $True \Rightarrow P(x) \vee R(x)$
3. $Q(y) \Rightarrow S(y)$
4. $R(z) \Rightarrow S(z)$

- **Resolution rule:-** In this step transitive implication is applied,

$$\frac{\alpha \Rightarrow \beta, \beta \Rightarrow \gamma}{\alpha \Rightarrow \gamma}$$

- Substitution of one predicate in the other. (i.e.) $P(x) \Rightarrow S(x)$ is substituted in $True \Rightarrow P(x) \vee S(x)$ that is instead of P(x), S(x) is substituted.
- Substitution of one predicate in the other. (i.e.) $R(x) \Rightarrow S(x)$ is substituted in $True \Rightarrow S(x) \vee R(x)$ that is instead of R(x), S(x) is substituted, which derives $True \Rightarrow S(x) \vee S(x)$ is equivalent to $True \Rightarrow S(x)$
- To prove using refutation, negation of given proof is added to the KB and derives a contradiction, then the given statement is true otherwise it is false.
- Therefore in step 4, we assume $S(A) \Rightarrow False$, derives a contradiction $True \Rightarrow False$.
- Therefore S(A) is true is proved using refutation technique in INF representation.
- The following diagram shows the proof that S(A) from KB1 using resolution with refutation in INF representation.



**D. Resolution with refutation using CNF representation:-**

**Given KB1:-**

1. ~P(w) ∨ Q(w)
2. P(x) ∨ R(x)
3. ~Q(y) ∨ S(y)
4. ~R(z) ∨ S(z)

- **Resolution rule:-**

$$\frac{\alpha \vee \beta, \beta \vee \gamma}{\alpha \vee \gamma} \quad \text{(i.e.)} \quad \frac{\sim P(w) \vee Q(w), \sim Q(w) \vee S(w)}{\sim P(w) \vee S(w)}$$

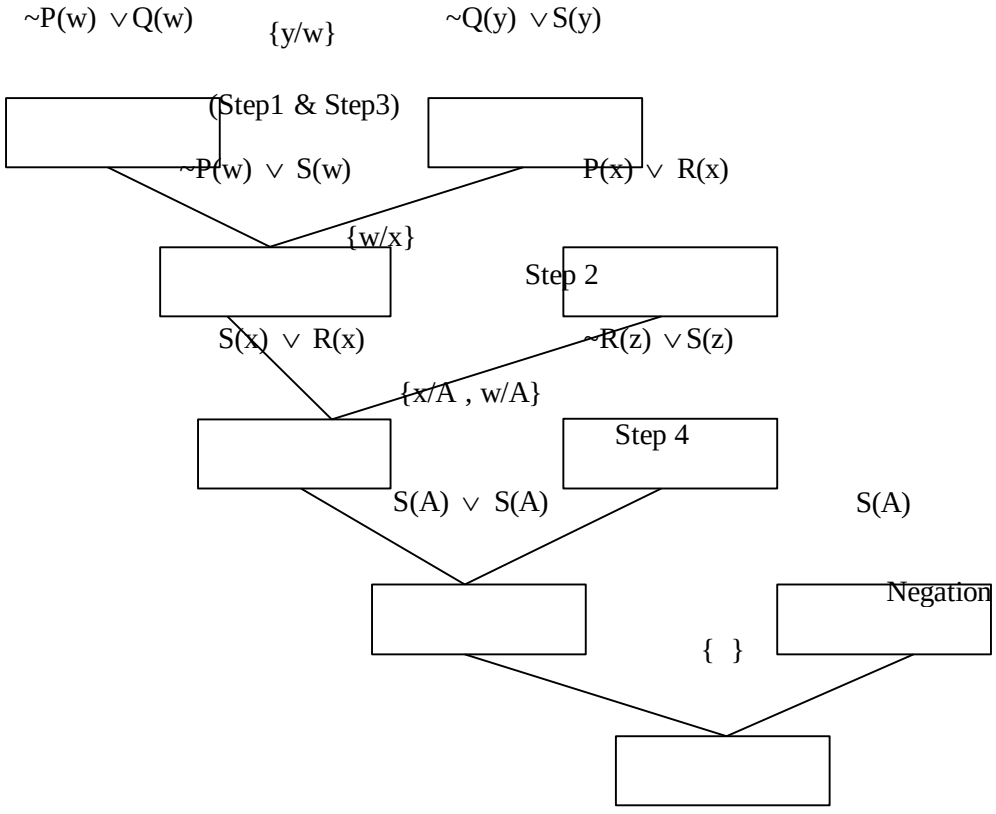
- **Resolution rule:-**

$$\frac{\alpha \vee \beta, \beta \vee \gamma}{\alpha \vee \gamma} \quad \text{(i.e.)} \quad \frac{R(x) \vee P(x), \sim P(x) \vee S(x)}{R(x) \vee S(x)}$$

- **Resolution rule:-**

$$\frac{\alpha \vee \beta, \beta \vee \gamma}{\alpha \vee \gamma} \quad \text{(i.e.)} \quad \frac{S(A) \vee R(A), \sim R(A) \vee S(A)}{S(A) \vee S(A)}$$

- To prove using refutation, the negation of the given statement to be proved is added to the KB and it derives a empty set, represents that the statement is True in the KB.
- Therefore S(A) is True is proved using resolution with refutation in CNF representation.
- The following diagram shows the proof that S(A) from KB1 using resolution with refutation in CNF representation.



### 2.10. 3 Resolution technique (From FOL Representation):-

- In the previous section we learned how to prove the given fact using two different types of resolution techniques (Resolution proof, Resolution with refutation proof) from the given KB representation (CNF, INF).
- Suppose if the KB is given as a English description, to prove a fact , then how to derive the conclusion? What are the sequence of steps to be done? Are discussed one by one,
  - The given KB description is converted into FOL Sentences
  - FOL sentences are converted to INF (or) CNF representation without changing the meaning of it (using conversion to normal form procedure)
  - Apply one of the resolution technique (Resolution proof (or) Resolution with refutation proof), to resolve the conflict.
  - Derive the fact to be proved or declare it as a incomplete solution.

#### 2.10.3.1 Conversion to Normal Form or Clause Form (Procedure) :-

1. **Eliminate Implications:-**
   Eliminate Implication by the corresponding disjunctions,
   (i.e.) $p \Rightarrow q$ is the same as $\neg p \vee q$
2. **Move $\neg$ inwards :-**
   Negations are allowed only on atoms in CNF and not at all in INF. Therefore eliminate negations with, Demorgan's laws, the quantifier equivalences and double negation.
   **Demorgans Law**
   $\neg(p \vee q)$ becomes $\neg p \wedge \neg q$
   $\neg(p \wedge q)$ becomes $\neg p \vee \neg q$
   **Quantifier equivalences**
   $\neg \forall x p$ becomes $\exists x \neg p$
   $\neg \exists x p$ becomes $\forall x \neg p$
   **Double Negation**
   $\neg\neg p$ becomes p Double negation
3. **Standardize variables:-**
   If the sentence consists of same variable name twice, change the name of one of the variable. This avoids confusion later when we drop the quantifiers,
   (i.e.) $(\forall x p(x)) \vee (\exists x Q(x))$ is changed into $(\forall x p(x)) \vee (\exists y Q(y))$
4. **Move quantifiers left:-**
   The quantifiers in the middle of the sentence to be moved to the left, without changing the meaning of the sentence
   (i.e.) $p \vee \forall$ xq becomes $\forall x \quad p \vee q$, which is true because p here is guaranteed not to contain x.
5. **Skolimize:-**

   Skolimization is the process of removing existential quantifiers by elimination, it is very similar to the existential elimination rule.
   (i.e.) $\exists x p(x)$ into p(A), where A is a constant that does not appear elsewhere in the KB.

**For Example:-** "Everyone has a heart"

- FOL : $\forall x$ person(x) $\Rightarrow$ $\exists y$ Heart (y) $\wedge$ Has (x, y)
- Replace $\exists y.....y$ with a constant H

$\forall x$ person (x) $\Rightarrow$ Heart (H) $\wedge$ Has (x, y)

- The above representation says that everyone has the same heart H, not necessarily shared between each other. It can be rewritten by applying a function to each person that maps from person to heart.

  $\forall x$ person (x) $\Rightarrow$ Heart (F(x) $\wedge$ Has (x, F(x)) where F is a function that does not appear elsewhere in the KB and it is called as Skolem function.

6. **Distribute $\wedge$ over $\vee$ :-**

   $(a \wedge b) \vee c$ becomes $(a \vee c) \wedge (b \vee c)$

7. **Flatten nested conjunctions and disjunctions :-**

   $(a \vee b) \vee c$ becomes $(a \vee b \vee c)$

   $(a \wedge b) \wedge c$ becomes $(a \wedge b \wedge c)$

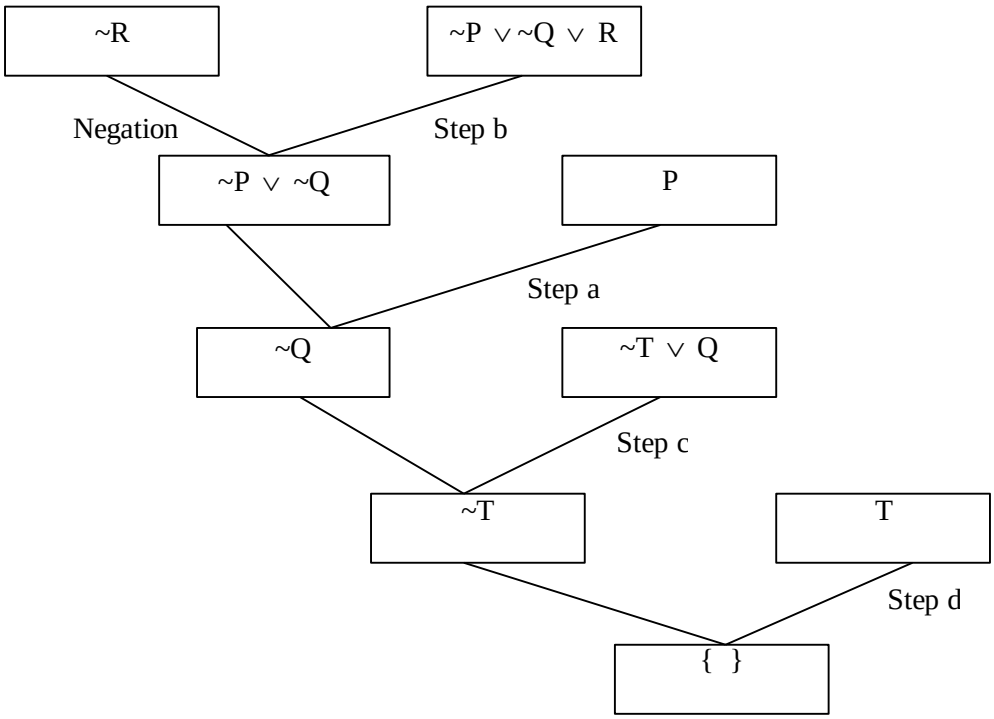
   It is a conjunction where every conjunct is a disjunction of literals.

8. **Convert disjunctions to implications :-**

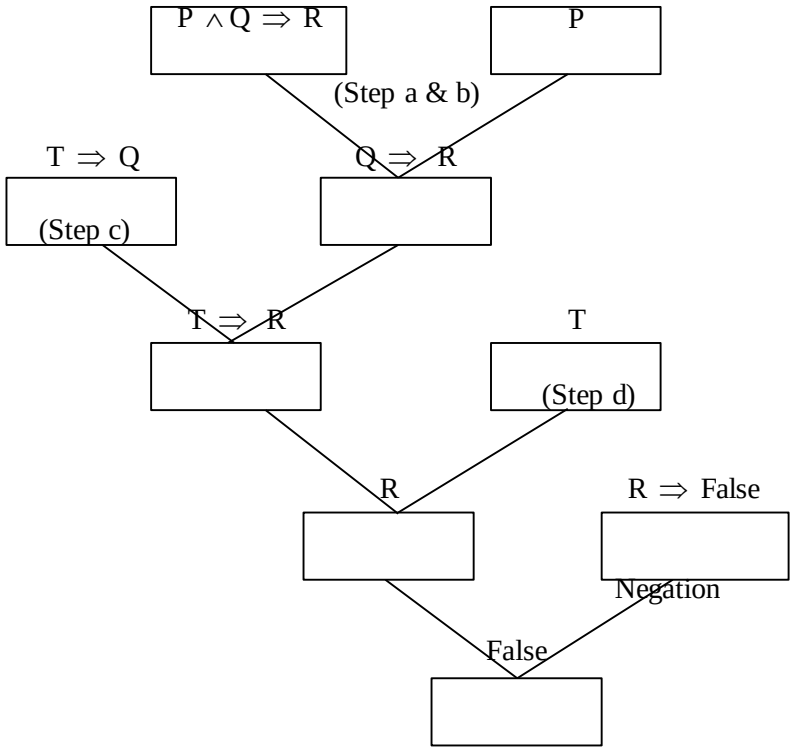
   From the CNF it is possible to derive the INF, (i.e.) combine the negative literals into one list, the positive literals into another and build an implication from them,

   (i.e.) $(\neg a \vee \neg b \vee c \vee d)$ becomes $(a \wedge b \Rightarrow c \vee d)$

- 1**. For Example: -** Covert the given axioms into equivalent clasuses form and prove that R is true using CNF and INF representation.
  - Axioms:
    - P
    - P $\wedge$ Q $\Rightarrow$ R
    - S $\vee$ T $\Rightarrow$ Q
    - T
  - Proof:
    - ✓ Equivalent Conjunctive Clause Form Representation:
      - P
      - $\neg$ P $\vee$ $\neg$ Q $\vee$ R
      - $\neg$ S $\vee$ Q, $\neg$ T $\vee$ Q
      - T
    - ✓ Equivalent INF Representation:
      - P
      - P $\wedge$ Q $\Rightarrow$ R
      - S $\Rightarrow$ Q, T $\Rightarrow$ Q
      - T
- The negation of the given statement (~R) derives a contradiction ({ }).
- Therefore it is proved that R is true.
- The following diagram shows the proof of resolution with refutation using CNF.



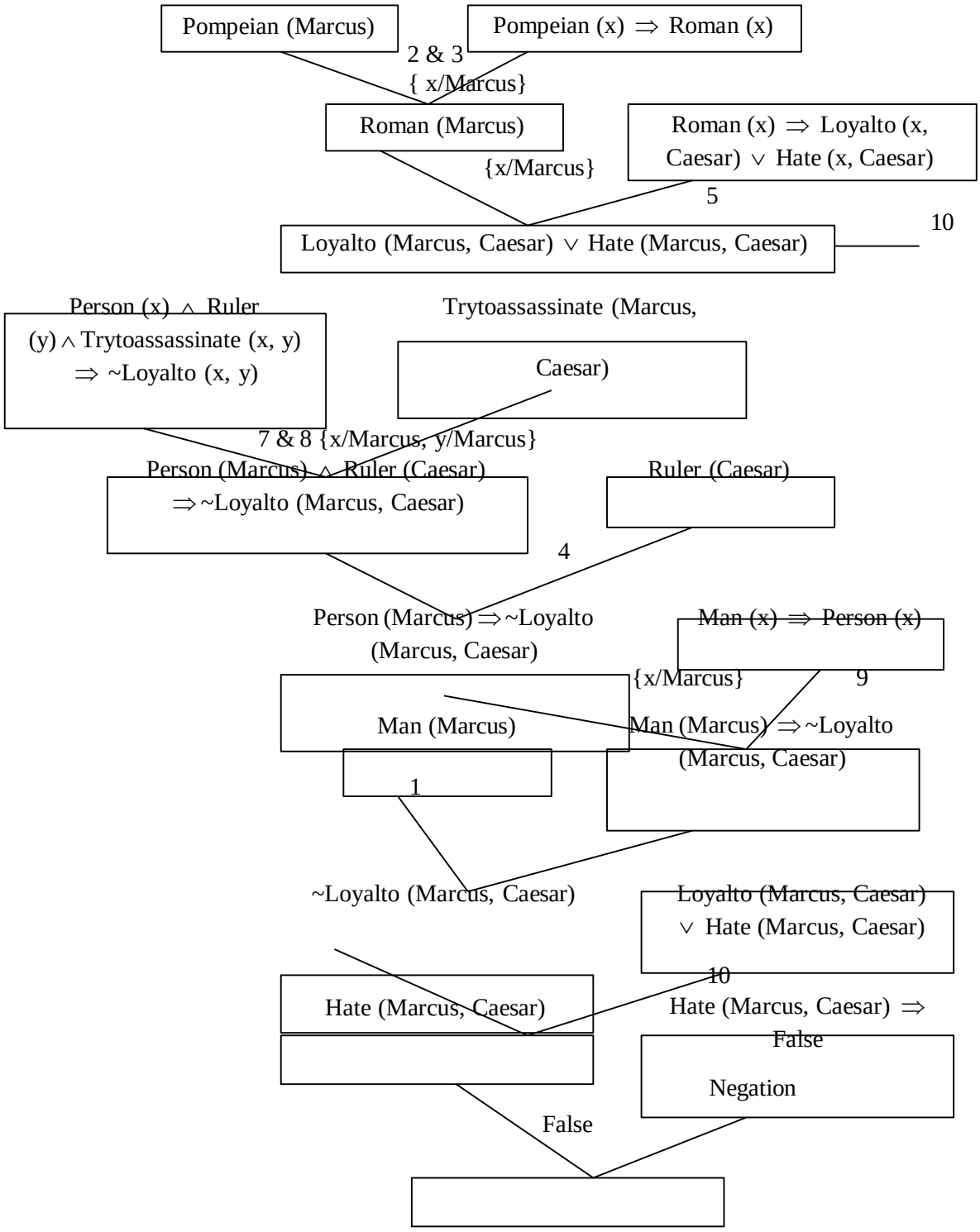

- The negation of the given statements (R ⇒ False) derives a contradiction (True ⇒ False).
- Therefore it is proved that R is True (i.e. Taken assumption is wrong)
- The following diagram shows the proof of Resolution with refutation using INF.



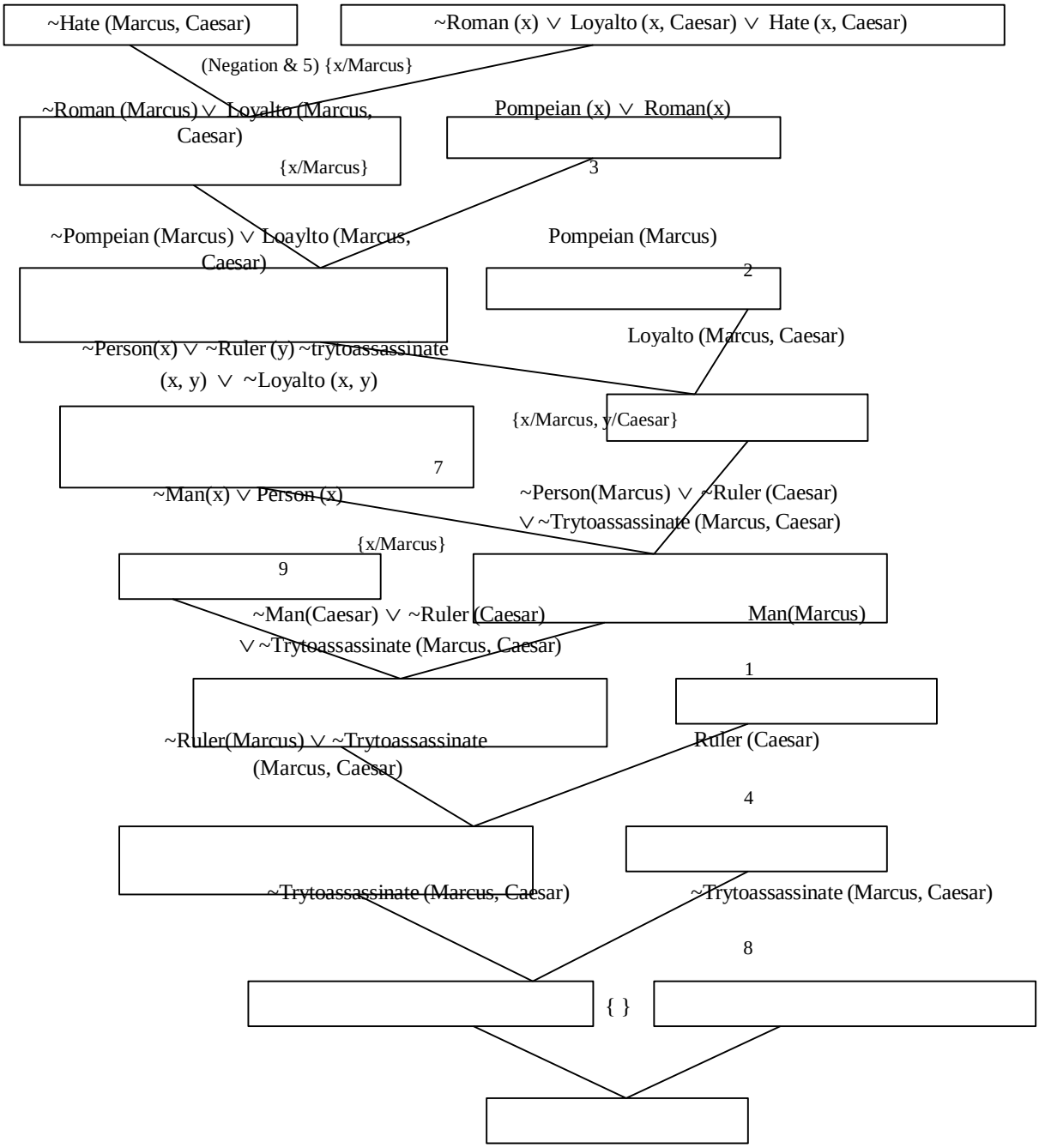
- **2. For Example: -** Convert the given sentences (KB2) into its equivalent FOL representation and then convert it into its CNF and INF representation and solve the task using resolution with refutation proof.

    - Given KB2:-
        - Marcus was a man
        - Marcus was a Pompeian
        - All pompeians were romans
        - Caesar was a ruler
        - All romans were either loyalto Caesar or hated him
        - Everyone is loyal to someone
        - People only try to assassinate rulers they are not loyal to.
        - Marcus tried to assassinate Caesar
        - All man are people
    - **Task: -** Did Marcus hate Caesar?
    - **Solution:-**
        - ✓ **FOL Representation:**
            1. Man (Marcus)
            2. Pompeian (Marcus)
            3. $\forall x$ Pompeian (x) $\Rightarrow$ Roman (x)
            4. Ruler (Caesar)
            5. $\forall x$ Roman (x) $\Rightarrow$ Loyalto (x, Caesar) $\vee$ Hate (x, Caesar)
            6. $\forall x \exists y$ Loyalto (x, y)
            7. $\forall x \forall \mathbf{y}$ Person(x) $\wedge$ Ruler(y) $\wedge$ Trytoassassinate(x, y) $\Rightarrow \neg$ Loyalto(x, y)
            8. Trytoassassinate (Marcus, Caesar)
            9. $\forall x$ Man (x) $\Rightarrow$ Person (x)
        - ✓ **INF Representation:**
            1. Man (Marcus)
            2. Pompeian (Marcus)
            3. Pompeian (x) $\Rightarrow$ Roman (x)
            4. Ruler (Caesar)
            5. Roman (x) $\Rightarrow$ Loyalto (x, Caesar) $\vee$ Hate (x, Caesar)
            6. Loyalto (x, y)
            7. Person(x) $\wedge$ Ruler(y) $\wedge$ Trytoassassinate(x, y) $\Rightarrow \neg$ Loyalto(x, y)
            8. Trytoassassinate (Marcus, Caesar)
            9. Man (x) $\Rightarrow$ Person (x)
        - ✓ **CNF Representation :**
            1. Man (Marcus)
            2. Pompeian (Marcus)
            3. ~Pompeian (x) $\vee$ Roman (x)
            4. Ruler (Caesar)
            5. ~Roman (x) $\vee$ Loyalto (x, Caesar) $\vee$ Hate (x, Caesar)
            6. Loyalto (x, y)
            7. ~Person(x) $\vee$~Ruler(y) $\vee$~Trytoassassinate(x, y) $\vee$~Loyalto(x, y)
            8. Trytoassassinate (Marcus, Caesar)
            9. ~Man (x) $\vee$ Person (x)

- To prove the given fact, we assumed the negation of it (i.e.) Hate(Marcus, Caesar) ⇒ False and using inference rules we derive a contradiction False, which means that the assumption must be false, and Hate(Marcus, Caesar) is True.
- Therefore it is proved that Hate (Marcus, Caesar) is True using INF Representation.
- The following diagram shows the proof of Resolution with refutation using INF representation.



- To prove the given fact, we assumed that the negation of it (i.e.) ~ Hate (Marcus, Caesar) and using inference rules, we derive a contradiction as empty set, which means that the assumption must be false, and Hate (Marcus, Caesar) is true in the given KB.
- Therefore it is proved that Hate (Marcus, Caesar) is true using CNF representation.
- The following diagram shows the proof of Resolution with refutation using CNF representation.



### .10.4 Dealing with equality:-

- The unification algorithm is used to find a equality between variables with other terms (i.e.) p(x) unifies with p(A), but p(A) and p(B) fails to unify, even if the sentence A=B in the KB.
- The unification does only a syntactic test based on the appearance of the argument terms, not a true semantic test (meaning) based on the objects they represent.
- To solve this problem, one of the way is, the properties can be followed as,

1. $\forall x$ $x = x$ - Reflexive
2. $\forall x, y$ $x = y \Rightarrow y = x$ - Symmetric
3. $\forall x, y, z$ $x = y \wedge y = z \Rightarrow x = z$ - Transitive
4. $\forall x, y$ $x = y \Rightarrow (p_1(x) \Leftrightarrow p_1(y))$ $x = y$, if the predicate name

   $\forall x, y$ $x = y \Rightarrow (p_2(x) \Leftrightarrow p_2(y))$ and arguments are same.
5. $\forall w, x, y, z$ $w = y \wedge x = z \Rightarrow (F_1(w, x) = F_1(y, z))$

   $w = y \wedge x = z \Rightarrow (F_2(w, x) = F_2(y, z))$

- The other way to deal with equality is **demodulation rule**. For any terms x, y and z where UNIFY (x, z) = θ , defined as :

$$\frac{x = y, (.....z.....)}{(.....SUBST(\theta, y).....)}$$

- A more powerful rule named paramodulation deals with the case where we do not know $x = y$, but we do know $x = y \vee p(x)$.

### 2.10.5 Resolution strategies:-

- A strategy which is used to guide the search towards a proof of the resolution inference rule. Different types of strategies are as follows,

  ✓ **Unit Preferences:-**

  This strategy prefers a sentence with single literal, to produce a very short sentence

  **For Example:-** P and $P \wedge Q \Rightarrow R$ derives the sentence $Q \Rightarrow R$.

  ✓ **Set of support:-**

  A subset of the sentence are identified (set of support) and resolution combines a set of support with another sentence and adds the resolvent into the set off support, which reduces the size of a sentence in the KB.

  **Disadvantage: -** Bad choice for the set of support will make the algorithm incomplete.

**Advantage: -** Goal directed proof trees are generated

✓ **Input resolution:-**

This strategy combines one of the input sentences (from the KB or the query) with some other sentence.

**For Example:-** Resolution proof, Resolution with refutation proof.

Input resolution is complete for KB that are in Horn form, but incomplete in the general case.

✓ **Linear resolution:-**

Two predicates are resolved (P and Q), if either P is in the original KB or if P is an ancestor of Q in the proof tree.

✓ **Subsumption:-**

A strategy which eliminate all sentences that are more specific than the existing sentence.

**For example:-** if P(x) is in the KB, then there is no sense in adding P(A) to KB.

### 2.10.6 Theorem Provers (or) Automated reasoners

- Theorem provers use resolution or some other inference procedure to prove sentences in full first order logic, often used for mathematical and scientific reasoning tasks.
- For Example:- MRS, LIFE.

| **Logic Programming language (PROLOG)** | **Theorem Provers** |
|---|---|
| Handles only the horn clauses | Handles full FOL representation |
| Choice of writing sentences in different form with same meaning affects the execution order | Does not affects the execution order derives the same conclusion |
| Example:- writing $A \leftarrow B \wedge C$ instead of $A \leftarrow C \wedge B$ affects the execution of the program | Example:- User can write either $A \leftarrow B \wedge C$ or $A \leftarrow C \wedge B$ or another form $\neg B \leftarrow C \wedge \neg A$ and the results will be exactly same. |

**Design of a Theorem Prover:-**

- An example for theorem prover is: OTTER=> Organized Techniques for Theorem proving and Effective Research, with particular attention to its control strategy.
- To prepare a problem for OTTER, the user must divide the knowledge into four parts;
  1. **SetOfSupport (SOS):** A set of clauses, which defines the important facts about the problem. Resolution step resolves the member of SOS with another axiom.

2. **Usable axioms: -** A set of axioms that are outside the set of support, provides background knowledge about the problem area. The boundary between SOS and the axiom is up to the user's judgement.
3. **Rewrites (or) Demodulators :-** A set of equations evaluated or reduced from left-to-right order (i.e.) x + 0 = x in which x + 0 should be replaced by the term x.
4. A set of parameters and clauses that defines the control strategy that is,
    a. Heuristic function :- to control the search
    b. Filtering function :- eliminates some subgoals which are uninteresting

**2.10.7 Execution:-**

- Resolution takes place by combining the element of SOS with useful axiom, generally prefers unit strategy.
- The process continuous until a reputation is found or there are no more clauses in the SOS.
- The following shows the algorithm of execution,

**Algorithm:-**

```
Procedure OTTER(sos, usable)

    Inputs: sos, a set of support-clauses defining the problem (global variable)

            Usable background knowledge potentially relevant to the problem
    Repeat:
            clause          the lightest member of sos
                    ←move clause from sos to usable
                        PROCESS (INFER(clause, usable),sos)
     Until sos = [] or a refutation has been found
function INFER(clause, usable) returns clauses
            resolve clause with each member of usable
            return the resulting clauses after applying FILTER

Procedure PROCESS (clauses, sos)
    for each clause in clauses do
        clause          SIMPLIFY (clause)
        merge identical literals
        discard clause if it is a tautology
        sos         [clause | sos]
        if clause has no literals then a refutation has been found
        if clause has one literal then look for unit refutation
    end
```

**2.10.8 Extending Prolog:-**

- Theorem prover can be build using prolog compiler as a base and a sound complete reasoned of full first order logic is done using PTTP.
- Where PTTP is Prolog Technology Theorem Prover.

- PTTP includes five significant changes to prolog to restore the completeness and expensiveness.

| Prolog | PTTP |
|---|---|
| Depth First Search | Iterative deepening search |
| Not possible like a PTTP implementation | Negated literals are allowed (i.e.) in the implementation P, ¬ P can be derived using two separate routines |
| Locking is not allowed | Locking is allowed (i.e.) A clause with n atoms is stored as n different rules. Example:- $A \Leftarrow B \wedge X$ is equivalent to $\neg B \Leftarrow X \wedge \neg A, \neg X \Leftarrow B \wedge \neg A$ |
| Inference is not complete | Inference is complete since refutation is allowed |
| Unification is less efficient | Unification is more efficient |

**Drawback of PTTP:-**

- Each inference rule is used by the system both in its original form and contrapositive form
  Example:- $(f(x, y) = f(a, b)) \Leftarrow (x = a) \wedge (y = b)$
- Prolog proves that two f terms are equal, But PTTP would also generate the contrapositive
  $(x \neq a) \Leftarrow (f(x, y) \neq f(a, b)) \wedge (y = b)$
- This is a wasteful way to prove that any two terms x and a are different.